Software Design in-the-small

Sandro Morasca

Università degli Studi dell'Insubria

Dipartimento di Scienze Teoriche e Applicate

Via Ottorino Rossi 9 – Padiglione Rossi

21100 Varese

sandro.morasca@uninsubria.it

Design-in-the-small

- Development activity whose result is to describe the parts of a software system
  - still in abstract terms, but
  - ready to be implemented without further design
- General (development) goals
  - high quality
  - low effort
  - short time
- Decisions, decisions, decisions …
  - data
  - methods/functions
  - policies

Problem 1: Software Modifiability

- A set of integer numbers can be represented in several different ways

```
public class SetOfIntegers
{
  public final static int SIZE = 10;
  public int n;
  public int list[] = new int [SIZE];
```

This is only an example …

Data should NOT be public!

Problem 1: Software Modifiability

- All applications that use `SetOfIntegers` are implemented based on this representation
- Suppose `SetOfIntegers` was written by a developer and other developers want to use it in their applications, which deal with
  - license plate numbers
  - student ID numbers
  - bank account numbers
  - …

Problem 1: Software Modifiability


```
public class LicensePlateManagement
{
  public SetOfIntegers setOfPlateNumbers;
  ...
  public int searchPlate(int plateNumber)
  {
    int i;
    while (i < setOfPlateNumbers.n &&
           plateNumber != setOfPlateNumbers.list[i] )
    {
      i++;
    }
    if (plateNumber != setOfPlateNumbers.list[i] )
    { return -1; }
    else
    { return i; }
  }
  ...
}
```

Problem 1: Software Modifiability


```
public class StudentIDNumberManagement
{ public SetOfIntegers setOfIDNumbers;
  ...
  public int readIDNumber() //read a number input by user
  { int readNumber;
    ...
    return readNumber;
  }
  public int searchIDNumber( int IDNumber) { ... }
  public void addIDNumber()
  { int number = readIDNumber();
    int position = searchIDNumber(number);
    if( position == -1 )
    {setOfIDNumbers.list[setOfIDNumbers.n] = IDNumber;
     setOfIDNumbers.n++; }
    ...
  }
  ...
}
```

Problem 1: Software Modifiability


```
public class BankAccountManagement
{
  public SetOfIntegers setOfAccountNumbers;
  ...
  public int searchAccount(int accountNumber)
  { ... }

  public void deleteAccount(int accountNumber)
  {
    int position = searchAccount(accountNumber);
    if (position != -1)
    {
      setOfAccountNumbers.list[position] = 0;
    }
  }
  ...
}
```

Problem 1: Software Modifiability

- Suppose the data structure of `SetOfIntegers` needs to be modified because
  - several faults were found in some of its methods, or
  - its objects take up too much memory space, or
  - having a one-size-fits all length for the inner list does not really fit clients classes' needs, or
  - …
- The meaning of operations, instead, must remain unchanged
- It will be inevitable to change all client classes
  - even though they are only interested in the functionalities of the `SetOfIntegers` class

Problem 1: Software Modifiability

New data structure

```
public class IntegerListNode
{
  public int item;
  public IntegerListNode next;
  ...
}

public class SetOfIntegers
{
  public IntegerListNode firstNode;
  ...
}
```

Problem 1: Software Modifiability


```
public class LicensePlateManagement
{
  public SetOfIntegers setOfPlateNumbers;
  ...
  public IntegerListNode searchPlate(int plateNumber)
  {
    IntegerListNode ref = setOfPlateNumbers.firstNode;
    while (ref != null &&
      plateNumber != ref.item )
    {
      ref = ref.next;
    }
    return ref;
  }
  ...
}
```

Problem 1: Software Modifiability


```
public class StudentIDNumberManagement
{ public SetOfIntegers setOfIDNumbers;
  ...
  public int readIDNumber() { }
  public IntegerListNode searchIDNumber(
                    int IDNumber) { ... }
  public void addIDNumber()
  {
    int number = readIDNumber();
      IntegerListNode ref = searchIDNumber(IDNumber);
    if( ref != null )
    {
      IntegerListNode newNode =
                          new NodoListaInteri(IDNumber);
      newNode.next = setOfIDNumbers.firstNode;
      setOfIDNumbers.firstNode = newNode;
    }
  }
  ...
}
```

Problem 1: Software Modifiability


```
public class BankAccountManagement
{
  public SetOfIntegers setOfAccountNumbers;
  ...
  public IntegerListNode searchAccount(int accountNumber)
  { ... }

  public void deleteAccount(int accountNumber)
  {
    IntegerListNode ref = searchAccount(accountNumber);
    if (ref != null)
    {
      <find the node before the one with accountNumber>
     prec.next = ref.next;
    }
  }
  ...
}
```

Problem 1 Software Modifiability

- It will be necessary to change
  - all of the statements that reference the old data structure
  - only the statements that reference the old data structure
- This is due to the fact that changes in the data structure imply changes in the interface of the class
- Problems
  - waste of time and effort
  - potential introduction of new faults

Problem 2: Data Integrity

- If several functions can freely access the data of a class, there is a chance that there will be problems when they
  - **use** the values of the data
  - **change** the values of the data

Problem 2: Data Integrity


```
public class SetOfIntegers
{
  public final static int SIZE = 10;
  public int n;
  public int list[] = new int [SIZE];
}
```

Problem 2: Data Integrity


```
public class BankAccountManagement
{
   ...
   public int searchAccount(int accountNumber,
                            SetOfIntegers setOfAccountNumbers)
   { ... }

   public void deleteAccount(int accountNumber,
                             SetOfIntegers setOfAccountNumbers)
   {
      int position = searchAccount(accountNumber,
                                   setOfAccountNumbers);
      if (position != -1)
      {
         setOfAccountNumbers.list[position] = 0;
      }
   }
   ...
}
```

Conventionally using value 0
to mean that an account does not exist

Problem 2: Data Integrity


```
public class SpecialAccountManagement
{
  public int searchAccount(int accountNumber,
                           SetOfIntegers setOfAccountNumbers)
  { ... }
  public void deleteAccount(int accountNumber,
                            SetOfIntegers setOfAccountNumbers)
  {
    int position;
    position = searchAccount(accountNumber,
                             setOfAccountNumbers);
    if (position != -1)
    {
      setOfAccountNumbers.n--;
      setOfAccountNumbers.list[position] =
        setOfAccountNumbers.list[setOfAccountNumbers.n];
    }
  }
  ...
}
```

Other way of deleting

Problem 2: Data Integrity


```
public class ForeignAccountManagement
{
  public int searchAccount(int accountNumber,
                           SetOfIntegers setOfAccountNumbers)
  { ... }
  public void deleteAccount(int accountNumber,
                            SetOfIntegers setOfAccountNumbers)
    int position;
    position = search(number);
    if (position != -1)
    {
      setOfAccountNumbers.list[position] =
        setOfAccountNumbers.list[setOfAccountNumbers.n];
    }
  }
  ...
}
```

Different deletion method, but it is inconsistent with the others

Problem 2: Data Integrity


```
public class AccountManagement
{
  SetOfIntegers setOfAccountNumbers;
  BankAccountManagement bank;
  SpecialAccountManagement special;
  ForeignAccountManagement foreign;

  public static void main()
  {
    int number;
    ...
    bank.delete(accountNumber, accountNumberSet);
    special.delete(accountNumber, accountNumberSet);
    foreign.delete(accountNumber, accountNumberSet);
}
```

Inconsistent use of data

Problem 3: Software Reuse

- So many different representations …
  - … just one type, though
- An abstract data type
  - methods must have the same behavior independent of the representation
- A data type that can be
  - reused by several applications without any modifications

Solution: Use a Module

- Separate
  - content
    - the data to access/the algorithms used
  - interface
    - the available ways to access the data/use the algorithms
- Further benefits, thanks to reuse
  - greater reliability
  - lower cost
  - lower development time

Example: Set of Integers


```
public class SetOfIntegers
{
  private final static int SIZE = 10;
  private int n;
  private int list[] = new int [SIZE];

  private int search(int number)
  {
    int i = 0;
    if ( isEmpty() )
    { return -1; }
    while ( (i < n -1) && (number != list[i]) )
    { i++;}
    if ( number != list[i] )
    { return -1; }
    else
    { return i; }
  }
```

Example: Set of Integers


```
public SetOfIntegers()
{
  n = 0;
}

public SetOfIntegers( SetOfIntegers otherSet )
{
  copy( otherSet );
}

public void finalize()
{
  n = 0;
}
```

Example: Set of Integers


```
public boolean isEmpty()
{
  return n == 0;
}

public boolean isFull()
{
  return n == SIZE;
}

public int cardinality()
{
  return n;
}

public boolean belongs(int number)
{
  return search(number) != -1;
}
```

Example: Set of Integers


```
public void insert(int number)
{
  if (!isFull() && (!belongs(number)))
  {
    list[n] = number;
    n++;
  }
}

public void delete(int number)
{
  int position;
  position = search(number);
  if (position != -1)
  {
    n--;
    list[position] = list[n];
  }
}
```

Example: Set of Integers


```
public boolean contains( SetOfIntegers otherSet )
{
  for (int i = 0; i < otherSet.n; i++)
  {
    if (!belongs(otherSet.list[i]))
    {
      return false;
    }
  }
  return true;
}

public boolean equals( SetOfIntegers otherSet )
{
  return contains( otherSet ) && otherSet.contains(
this );
}
```

Example: Set of Integers


```
public void emptySet()
{
  n = 0;
}

public void copy( SetOfIntegers otherSet )
{
  n = 0;
  for (int i = 0; i < otherSet.n; i++)
  {
    list[n] = otherSet.list[i];
    n++;
  }
}
```

Example: Set of Integers


```
 public void union( SetOfIntegers otherSet )
{
  for (int i = 0; i < otherSet.n; i++)
  {
    if (isFull())
    { return; }
    insert(otherSet.list[i]);
  }
}

public void difference( SetOfIntegers otherSet )
{
  for (int i = 0; i < otherSet.n; i++)
  {
    if (isEmpty())
    { return; }
    delete(otherSet.list[i]);
  }
}
```

Example: Set of Integers


```
public void intersection( SetOfIntegers otherSet )
 {
   int i = 0;

   while(i < n)
   {
     if (!otherSet.belongs(list[i]))
     {
       delete(list[i]);
     }
     else
     {
       i++;
     }
   }
 }
```

Example: Set of Integers


```
public String toString()
  {
    String result = "";

    for (int i = 0; i < n; i++)
    {
      result += (list[i] + "\n");
    }

    return result;
  }
}
```

Modules

- A module is a part of a system that provides a set of services to other modules
- Services are computational elements that other modules may use

Sample Types of Modules

- Abstract operation
  - method/function
- Abstract object (abstract state machine)
    - e.g. a ***single*** set of integers module
  - a module that encapsulates a data structure
  - exports a set of operations
  - application of operation changes the state of the encapsulated object
- Abstract data type
  - a module that allows abstract objects to be instantiated

Module Interface

- The set of services provided by a module (exported) constitutes the module's interface
- The interface defines a contract between the module and its users
- A module consists of its interface and its body (implementation, secrets)
- Users only know a module through its interface

Sort Functions in C++


```
//Insertion Sort
void sort(int array[], int n) {
    int i, j, temp;
    for i = 1; i < n; i++) {
        j = i;
        while(j > 0 && array[j-1] > array [j]) {
            temp = array[j];
            array[j] = array[j-1];
            array[j-1] = temp;
            j--;
        }
    }
}
```

Sort Functions in C++


```
//Bubble Sort: same signature
void sort(int array[], int n) {
    int i, j, temp;
    for(i = 1; i < n; i++) {
        for(j = 0; j < n-i; j++) {
            if(array[j] > array[j+1]) {
              temp = array[j];
              array[j] = array[j+1];
              array[j+1] = temp;
            }
        }
    }
}
```

Class and Instance Members

- Classes and instances (objects) have
  - member data
  - member methods
- External data and methods are called non-members

- Instance members
  - there exists an independent copy for each instance
- Class members
  - there exists just one copy, shared by all class instances
    - **`static`** members in Java

Public Members

- Public members are
  - data (**NOT RECOMMENDED**)
  - methods

  that are visible to all classes and their instances

- Qualified by **`public`** keyword in Java

- Encapsulation is typically used to hide data and allow access only via the functionalities made public by the programmer
  - the **`public`** section should not include member data

Protected Members

- Protected members of a class in Java can be accessed only by members
  - of the class and its instances
  - of the subclasses and their instances
  - of the classes in the same package as the class
- Qualified by the **`protected`** keyword in Java
- A differenza degli elementi di una classe/un oggetto con visibilità limitata al package, viene esplicitata l'intenzione di rendere questi elementi riutilizzabili dalle sottoclassi che ereditano dalla classe

Package-private Members

- Package-private members are
  - data (**NOT RECOMMENDED**)
  - methods

  that are visible to all classes in the package and their instances

- This is the default visibility
  - no qualifier keywords used in Java

Private Members

- Private members are
  - data (best visibility for data)
  - methods

  that are not visible from outside the class and its instances

- Qualified by the **`private`** keyword in Java

`friend` Functions (Methods) in C++

- A class can allow a **`friend`** function to directly access its non-**`public`** members

```
class Rectangle {
    int width, height;
  public:
    Rectangle() {}
    Rectangle (int x, int y) : width(x), height(y) {}
    int area() {return width * height;}
    friend Rectangle duplicate (const Rectangle&);
};
```

`friend` Functions (Methods) in C++

- **`friend`** function

```
Rectangle duplicate (const Rectangle& param)
{
  Rectangle res;
  res.width = param.width*2;
  res.height = param.height*2;
  return res;
}
```

`friend` Functions (Methods) in C++

- Example of use of a **`friend`** method

```
int main () {
  Rectangle foo;
  Rectangle bar (2,3);
  foo = duplicate (bar);
  cout << foo.area() << '\n';
  return 0;
}
```

`friend` Classes in C++

- A class can allow another class to directly access its non-**`public`** members

```
class Rectangle {
    int width, height;
  public:
    int area ()
      {return (width * height);}
    void convert (Square a);
};
```

`friend` Classes in C++

- Class `Square` allows class `Rectangle` to access its **`private`** section

```
class Square {
  friend class Rectangle;
  private:
    int side;
  public:
    Square (int a) : side(a) {}
};
```

`friend` Classes in C++

- Example of use of a **`friend`** class

```
void Rectangle::convert (Square a) {
  width = a.side;
  height = a.side;
}
int main () {
  Rectangle rect;
  Square sqr (4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

`friend` Relationship in C++

- The **`friend`** relationship in C++
  - is not symmetrical
  - is not transitive
  - does not get inherited

Method Signature

- The signature of a method/function fully describes its external behavior
  - visibility
  - returned type
  - name
  - parameter list
  - exceptions
  - …

Method Overloading

- Overloading
  - giving the same token more than one meaning
- Overloaded methods may
  - differ in some elements of the signature, or
  - override existing methods

Operator overloading in C++

```
class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)  {real = r;   imag = i;}
    Complex operator + (Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print() { cout << real << " + i" << imag << endl; }
};
```

Operator Overloading


```
int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
```

- Output
  - `12 + i9`

In/out Parameters

- **in** parameters
  - the actual parameter's value is only passed to the method/function
    - when the method/function returns the actual parameter's value is unchanged
- **out** parameters
  - the actual parameter does not necessarily have a value before a method/function call
    - it takes one when the method/function returns
- **inout** parameters
  - the actual parameter's value is passed to the method/function
    - when the method/function returns the actual parameter's value may be changed

Names

- Using meaningful identifiers it important
  - design must convey the maximum amount of useful information
- Ambiguous/useless signature of a method

```
public float trapezoidArea(float a, float b, float c)
```

  - which parameters are the bases and which is the height?
  - need to write a comment
- Better signature (self-documenting)

```
public float trapezoidArea(
                    float base1, float base2, float height)
```

Exceptions

- Exceptions are a part of the behavior of a method
- A method must report on exceptional behaviors
  - those that can be somehow foreseen
  - only those that cannot/must not be dealt with in the method
- Example: a search method fails to find an element when invoked by a caller
  - should the search method not report the problem to the caller?
    - no, because the caller would assume that everything went fine
  - should the search method try to deal with that case?
    - no, because it does not know what the caller may want to do now

Cathegories of Member Methods

- Setter
  - Non-private member method that may change the values of member data
- Getter
  - Non-private member method that may not change the values of member data, but can only access them
- Utility
  - Private member method

- Methods can return an object of any class, including the class the method belongs to

`const` Methods in C++

- Getter methods can be qualified by the **`const`** keyword in C++
  - not mandatory, but useful even at design time

```
class Test {
    int value;
public:
   Test(int v = 0) {value = v;}
   int getSquaredValue() const {return value*value;}
// If this method contained a statement like
// "value = 100;" there would be a compile-time error
};
```

Constructors

- Different types of constructors exist
  - default constructor—at most one per class
  - constructor with parameters—as many as we like
    - copy constructor

- Which constructors should be provided?
  - In Java, including a constructor with parameters prevents the use of the default constructor of the superclass
  - In C++, it is important to define a copy constructor, used for
    - by-value parameter passing (in or out)
    - constructing an object based on another object of the same class
    - generating a temporary object when returning an object

Assignment and Equality

- They may have different meanings in different languages
  - shallow, involving only references
  - deep, involving objects

- It may be necessary to define assignment and equality methods that work as we intend, e.g.,
  - by overloading the assignment operator in C++
  - by redefining the `equals` method in Java

`static` Members

- These are class members
  - there is only one copy of them per class
- Typically qualified by **`static`**

```
class ClassIdentifier
{
   ...
   private static int numberOfObjects;
   ...
   private static int someInteger = 14;
   ...
   private static OtherClass object1;
   ...
   private static OtherClass object2 =
                                   new OtherClass();
   public static aMethod(int anotherInteger) {…}
   ...
}
```

`static` Member Data

- Member data
  - `numberOfObjects` initialized with 0
  - `someInteger` initialized with 14
  - `object1` initialized with **`null`**
  - `object2` initialized with the default constructor of `OtherClass`
- **`static`** data are allocated at the beginning of execution, independent of the existence of any objects of the class

`static` Members

- **`static`** members do not belong to any object, so they cannot use the **`this`** reference
- **`static`** methods can only use **`static`** members of the same class (in addition to **`static`** members of objects of other classes)
- It is advised to use them by their class identifier

```
ClassIdentifier.aMethod(3)
```

`final` Methods and Classes

- Keyword **`final`** qualifies
  - methods that cannot be redefined in the subclasses of a class
  - classes that cannot be extended via inheritance
- Examples

```
public final void method()
```

```
public final class SubClass extends SuperClass
```

- **`static`** methods are also **`final`**, so they cannot be redefined
- **`private`** methods can be redefined
  - they are not visible in subclasses anyway
  - so, **`private static`** methods can be redefined

Why Define Methods and Classes `final`?

- When we want to be absolutely sure nobody overrides the method or class and changes its semantics
- When you would rather use composition over inheritance
- For performance reasons: dynamic binding implies possible
  - delays
  - unkown execution times
- When objects must be immutable, like with `String`
- For security reasons: no ways around
- When there may be changes in the class that would imply breaking its subclasses

`abstract` Methods and Classes

- Keyword **`abstract`** qualifies
  - methods that must be redefined in the subclasses of a class
  - classes that must be extended via inheritance
- It is not possible to instantiate an **`abstract`** class
  - only objects of its non-**`abstract`** subclasses
- If at least one of its method is **`abstract`**, the entire class is **`abstract`**
  - in Java, it must be declared **`abstract`**
- Examples

```
public abstract void method()
```

```
public abstract class SubClass extends SuperClass
```

Interfaces

- Keyword **`interface`** indicates a class "skeleton" where
  - methods are all **`public abstract`**,
    - even if it is not explicitly written, and
    - they must be redefined in the classes that implement it
  - data are all **`public final static`**
    - even if it is not explicitly written


```
public interface Intf
{
    public final static int CONSTANT1 = 30;
    public double CONSTANT2 = 3.14;
      //it is final static anyway
    public abstract void m( int x );
    public int q( double y );
      //it is abstract anyway
 }
```

Genericity

- It is another design decision, e.g.,
  - set of integers, or
  - set of objects, or
  - set of some kind of objects?
- The inner functioning of the set operations are the same
  - regardless of the type
- Compare
  - reuse potential
  - additional effort

Policies

- A policy is a strategy used to implement methods and deal with data
  - it is a design decision
- Policies should be hidden
  - none of the clients' business
  - they can be replaced with other policies without changing the clients

Conventions

- Conventions must be avoided, to the extent possible
  - in any case, they should be confined to the smallest possible scope
- Example: a search method returns a negative number if the element searched does not exist
  - it can be used for a private method
  - it should not be used for a public method
- Another example: `String.compareTo` returns an integer
  - < 0 if the first string precedes the second
  - = 0 if the two strings are the same
  - > 0 if the first string follows the second

Design by Contract

- The visible programming interface of a module (e.g., a method, a class) defines a contract
  - an agreement between a *client* and a *contractor*
  - defines *obligations* to achieve *benefits*

Real-life Example: Hiring a Software Consultant

- Client
  - Obligations
    - payment of a specified amount, in various installments
    - access to proprietary information for requirements
  - Benefits
    - new software product
- Contractor
  - Obligations
    - delivery of software product by the specified deadline
    - non-disclosure agreement
  - Benefits
    - income

Contract for an OO Module

- Define what each method requires (obligation for the client)

  precondition

- and what each method provides (obligation for the contractor)

  postcondition

  Preconditions and postconditions may be expressed using logic

Example

- Operation `insert(int number)` in a `SetOfIntegers` instance `mySet`

Precondition

- no_elements < size, i.e.,
- `mySet.cardinality() < SIZE`

Postcondition

- all elements of `mySet` before `insert` also belong to it after running `insert` AND `number` belongs to `mySet` too, i.e., (the prime symbol denotes values after method execution)
- `mySet' contains mySet` AND `number` belongs to `mySet'`, i.e.,
- `mySet.insert(number).contains(mySet)` && `mySet.belongs(number)`

Which Preconditions?

- Should a routine be prepared to handle all possible inputs?
  - NO
    - WEAK precondition (TRUE means no constraints at all); all complications delegated to routine
    - STRONG precondition (FALSE means it cannot be invoked at all)
  - The choice of the precondition is a design decision; there is no absolute rule
    - preferable to write simple routines that satisfy a well-defined contract rather than a routine that tries to attempt every imaginable situation

Preconditions and Postconditions: Who Does What

- Precondition
  - The client must guarantee the property
    - If `p` is the precondition for method `m`, either we write (for any object `x`)

      ```
      if (x.p) x.m(…)
      else … special treatment
      ```

    - or we ensure that `p` holds before the call by reasoning on the program
- Postcondition
  - The contractor must guarantee it in the method's implementation

Internal Properties of a Class

- We can specify a property that all instances should satisfy as an *invariant*
  - $0 \leq$ no_elements $\leq$ size
- The invariant must be true
  - after creation
  - before and after each operation
- The invariant defines an additional obligation
  - the class implementation must satisfy it

Class Correctness

- Creation operation
  - $\{pre_c\}$ constructor $\{INV'\}$
- Any other operation
  - $\{pre_{operation} \wedge INV\}$ operation $\{post_{operation} \wedge INV'\}$

Preconditions

The client is responsible for their truth …however…

- it may be impossible to evaluate applicability before application of operation
  - overflow, input/output, …
- frequent operations which almost never fail
  - new and memory exhausted
- there are errors that cause invocations that do not satisfy the precondition

The Role of Exceptions

- They should be raised if one of the following conditions are violated
  - precondition
  - postcondition
  - invariant
- When control leaves a routine, either its postcondition and the invariant are true or an exception is returned
  - returned exceptions should be listed in the interface

Inheritance

- Subclasses can add attributes and methods
- Can redefine methods
  - syntactic constraints
    - covariance of result and countervariance of parameters
  - semantic constraints
    - $pre_{class} \rightarrow pre_{subclass}$
    - $post_{subclass} \rightarrow post_{class}$