Universitatea Tehnica din Cluj-Napoca
Departament Calculatoare

Programming Techniques in Java

Class Design I

2017

Contents

Separate Interface from Implementation

- When the class functionality can be implemented in different ways => separate the interface from the implementation
- Example
  - List implementation
    - Ordered collection - accessed through an index.
    - The size of the list can grow as needed
  - Implementation alternatives
    - Linked list
    - Dynamically resizable array
- Advantages
  - Hidden implementation details
  - Changes in implementation
- Interface
  - Described by List

Implementations

- LinkedList

```
public class LinkedList implements List
    // … implementation body
}
```

- DynamicArray

```
public class DynamicArray implements List
    // … implementation body
}
```

- Examples form JCF

Encapsulation

- Hide implementation details
  - Every class has implementation secrets that should be kept secret
- Constructors should build only valid objects
- Role of data visibility qualifiers
- Use of accessor (getters) and mutators (setters) to access implementation details
- Setters and getters code is controlled by class designer
  - The code should preserve the valid states of objects
  - Define setter and getter only if necessary
  - Don't automatically supply set methods for every instance field

Visibility

- Visibility: public, protected, (default), private
- Rule

  Assign fields and methods the most restrictive visibility possible while still providing the needed functionality
- **Note**. The more a class exposes its methods and fields to other classes

  => tightly coupling

  => difficult to maintain and modify without breaking other code

Encapsulation

Example

// non - encapsulated

```
public class Employee {
  public int employeeID;
  public String firstName;
  public String lastName;
}
// ... In a program we may have
Employee emp = new Employee();
emp.employeeID = 123456;
emp.firstName = "John";
emp.lastName = "Smith";
```

// encapsulation in action

```
public class Employee {
  // ... see the use of mutator and accessor methods
  private int employeeID;
  private String firstName;
  private String lastName;

  public int getEmployeeID() {
    return employeeID;
  }
  public void setEmployeeID(int id) {employeeID = id;}
  public String getFirstName() {return firstName;}
  public void setFirstName(String name) {
     firstName = name;
  }
  public String getLastName() {return lastName;}
  public void setLastName(String name) {
    lastName = name;
  }
}
```


Encapsulation

- Advantages of encapsulation
  - Thread – safe objects (due to private qualifier)
  - Accessors and mutators allow assigning only valid values to instance variables
  - Accessors and mutators insulate the class from changes to a property's implementation
- For example, you could change employeeID from an int to a String without affecting other classes, as long as you perform the appropriate conversions in the accessor and mutator methods

```
public class Employee {
  private String employeeID;
  private String firstName;
  private String lastName;

  public int getEmployeeID() {
    return Integer.parseInt(employeeID);
  }
  public void setEmployeeID(int id) {
    employeeID = Integer.toString(id);
  }
  ...
}
```

- employeeID type was changed
- The client classes may read / modify it without observing any change due to the encapsulation of employeeID (behind mutator and accessor methods)

Encapsulation
Side Effects

- Side effect of a method - any data modification that is observable when the method is called
- If a method has no side effects => when called always returns the same answer (provided, that no other methods with a side effect have been called in the meantime)
- A method may modify
  - The explicit parameter object (i.e. this object) which means that the method is a mutator method
  - Other objects using explicit parameters, or accessible static fields
- Functional programming – avoids side effects

Encapsulation
Side Effects

Example 1 - Side effect of a method through explicit parameter

- Consider
  - al1 and al2 as array lists and
  - a library method addAll

  al2.addAll(al1);
- The user expects that **al2** will be modified as a result but don't expects **al1** to be modified (as a side effect) by the method **addAll**

Example 2 - Side effect of a method through accessible static field (such as System.out in the example below)

```
// array based stack implementation
public void push() {
  if (stk.isFull())
  System.out.println ("Out of Space" );
  // Wrong approach
}
```

- Correct approach: Error condition should be reported by throwing an exception

Encapsulation
Law of Demeter

- Law of Demeter [Karl Lieberherr] - A method should only use
  - Instance fields of its class
  - Parameters
  - Objects that it constructs with new
- A method that follows the Law of Demeter does not operate on global objects or objects that are a part of another object
- In particular: a method should not ask another object to give it a part of its internal state to work on
- The Law of Demeter implies that a class should not return a reference to an object that is a part of its internal implementation.

Immutable Classes and Objects

- Mutable objects – state can change
- Immutable objects – state is set during construction process and never changes then
- Examples: wrapper classes Integer, Boolean, Double, class String (new string objects are actually generated)
- Benefits of immutable classes and objects
  - Immutable class is inherently thread-safe
  - i.e. Any number of threads can safely reference and use an immutable object, without any explicit synchronization between the other threads
- How to enforce a class to generate immutable objects
  1. Define its fields as **final**
  2. Remove mutator (setter) type methods
  3. Have one (or more) class constructor(s) with parameters that assign values to all instance variables

Immutable Classes and Objects

```
public class Employee {
  private String employeeID;
  private String firstName;
  private String lastName;

  public int getEmployeeID() {
    return Integer.parseInt(employeeID);
  }

// constructor
public Employee(String id, String first, String last) {
  ployeeID = id;
  rstName = first;
  lastName = last;
}

// should be removed
public void setEmployeeID(int id) {
    employeeID = Integer.toString(id);
  }
  ...
}
```

Using final

-The class below is similar to the one in the left
-tells you immediately that this is a immutable class
-otherwise (in the left) you have to look at all class definition to find out that it is immutable

```
public class Employee {
  private final String employeeID;
  private final String firstName;
  private final String lastName;

  public int getEmployeeID() {
    return Integer.parseInt(employeeID);
  }

// constructor
public Employee(String id, String first, String last) {
  ployeeID = id;
  rstName = first;
  lastName = last;
}

// should be removed
public void setEmployeeID(int id) {
    employeeID = Integer.toString(id);
  }
  ...
}
```


Immutable Classes and Objects

Example 1 – Non immutable class Student

- Having fields as private doesn't mean that the class is immutable
  - It may have a reference to an object and that object may be changed by other classes!!
  - See example below

```
public class Student {
  private List<TestScore> testScores;
  private String name;
  public Student(List<TestScore> scores,
        String name) {
    this.testScores = scores;
    this.name = name;
  }

  public List<TestScore> getTestScores() {
     return testScores;
  }

  //… all setter methods are removed
```

-getTestScores() method returns a reference to the list of scores => the scores could be modified

Note.

Sets and other collections returned from a method should be immutable to preserve encapsulation

Immutable Classes and Objects
Example 2 – Immutable class Student

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Student {
  private List<TestScore> testScores;
  private String name;
  public Student(List<TestScore> scores,
            String name) {
     this.testScores =
Collections.unmodifiableList (
         new ArrayList<TestScore>(scores));
         this.name = name;
  }
  public String getName() { return name;}
  public List<TestScore> getTestScores() {
     return testScores;
  }
  // …
}
```

- Advantage of classes without mutator methods:
  - Their object references can be freely shared.
- !! Pay attention when sharing mutable objects.
  - It is dangerous for an accessor method to give out a reference to a mutable instance field
- **Conclusion (after studying the example) – favor immutable objects whenever possible**

Immutable Classes and Objects

Example

```
class Employee {
 // instance variables
  private String name;
  private double salary;
  private Date hireDate;
  …
  public String getName() {return name;}
  public double getSalary(){return salary;}
  public Date getHireDate() {return hireDate;}
}
```

Immutable Classes and Objects

See example
- **Problem 1**
- **getHireDate** method breaks encapsulation
- Since the Date class is a mutable class, anyone can apply a mutator method to the returned reference and thereby modify the Employee object
- getName is safe because String class is immutable

```
// breaking the code above
Date d = vasile.getHireDate();
d.setTime(t) ; / / Changes vasile's state!

// fix the problem
public Date getHireDate() {
  return (Date) hireDate.clone( );
}
```

See example
- **Problem 2**
- Class constructor may brake encapsulation

```
public Employee (String aName,
                         Date aHireDate) {
  name = aName ;
  hireDate = aHireate;
}

// Breaking encapsulation
Date d = new Date() ;
Employee e = new Employee ("Vasile", d) ;
d.setTime ( . . . ) ;
```

- To fix the problem, clone the hire date in the constructor

Object equality

- Object identity-based equality
- Default object implementation
  - identity based
  - o1.equals(o2)
- Object state-based equality
  - Most classes should override equals method to implement a content based equality

Object equality

Method contract

- **Reflexivity**
  - for any non-null reference value x => x.equals(x) should return true
- **Symmetry**
  - for any non-null reference values x and y => x.equals(y) should return true if and only if y.equals(x) returns true
- **Transitivity**
  - for any non-null reference values x, y, and z =>
    if x.equals(y) returns true and
    y.equals(z) returns true,
    then x.equals(z) should return true.
- **Consistency**
  - for any non-null reference values x and y, multiple invocations of **x.equals(y)** consistently return true or consistently return false
    (provided no information used in equals comparisons on the objects is modified)
- **Non-nullity**
  - for any non-null reference value x, **x.equals(null)** should return false.

Note

- Overriding **hashCode** method
- **hashCode** general contract
  - equal objects must have equal hash codes.

Object equality

equals - skeleton

```
public class C {
 // ... class resources

  public boolean equals (Object o) {
    if (o == this) return true;
    if (!(o instanceof C)) return false;
    C cObj = (C) o;
    return ... ; // logical test of equality
  }
}
```

- Step 1. Use == operator to check if the argument is a reference to this object
- Step 2. Use **instanceof** operator to check if the argument is of the correct type.
- Step 3. Cast argument to the correct type
- Step 4. For each “significant” field in the class (see details, next slide)
  - check to see if that field of the argument matches the corresponding field of this object
- Step 5. At the end, ask yourself three questions:
  - is it symmetric,
  - is it transitive, and
  - is it consistent?

Object equality
equals - skeleton

Step 4 details

- For primitive fields p (other than float and double)
  if(p != o.p) return false;
- For float fields
  - use Float.floatToIntBits (translate to int values and compare ints using ==)
- For double fields
  - use Double.doubleToLongBits
- For object reference fields
  - invoke equals recursively;
- Some instance variables could contain null values
  - Avoid throwing NullPointerExceptions
  (field == null ? o.field == null : field.equals(o.field))
- Approaching the fields
  - temporarily fields,
  - derived fields (from other fields) or
  - non essential fields
  => excluded (not compared)

Object equality
equals -Example

```
public interface List {
  // mutators
  public void addElement(Object le, int i);
  public void addFirst(Object le);
  public void addLast(Object le);
  public Object remove(int i);
  public Object removeFirst();
  public Object removeLast();
  // getters (accessors)
  public Object getFirst();
  public Object getLast();
  public Object getElement(int i);
  public int getSize();
  // test
  public boolean isEmpty();
 // overrides
 public boolean equals (Object o);
}
```

```
public boolean equals(Object o) {
  if (this == o) return true;
  if (o instanceof LList) {
    LList oList = (LList) o;
    if(this.getSize() == oList.getSize()) {
      for(int i = 0; i < this.getSize(); i++) {
        Object thisItem = this.getElement(i);
        Object oItem = oList.getElement(i);
        if(thisItem == null) {
          if(oItem != null) { return false; }
        } else {
          if(!thisItem.equals(oItem)) {
            return false;
          }
        }
      } // for
      return true;
    } // if
  } // if
  return false;
}
```

Object equality
When not override equals

- Unique class instances
- Doesn't matter whether the class provides a “logical equality” test
- A superclass has already overridden equals
  – the behavior inherited from the superclass is appropriate for this class.
- The class is private or package-private
  – Only when you are certain that its equals method will not be invoked
  – Just in case protection:

```
public boolean equals(Object o) {
  throw new UnsupportedOperationException();
}
```


Object equality
Problems

- Example - classes Point, ColorPoint
  – Two Point objects are equal if they have the same position
  – Two ColorPoints are equal if they have the same position and color

```
public class Point {
  private final int x;
  private final int y;
  public Point(int x, int y) {
    this.x = x; this.y = y;
  }
  public boolean equals(Object o) {
    if (!(o instanceof Point) return false;
    Point p = (Point)o;
    return p.x == x && p.y == y;
  }
  // … rest of class resources
}
```

```
public class ColorPoint extends Point {
  private Color color;
  public ColorPoint(int x, int y, Color
     color) {
    super(x, y);
    this.color = color;
  }
  // … rest of class resources
  // method equals on following slides
}
```

Object equality
Problems

```
// Try 1
public boolean equals(Object o) {
   if (!(o instanceof ColorPoint) return false;
  ColorPoint cp = (ColorPoint)o;
  return super.equals(o) && cp.color == color;
}

Point p = new Point(1, 2);
ColorPoint cp = new ColorPoint(1, 2, Color.RED);
```


Object equality
Problems

```
// Try 1
public boolean equals(Object o) {
   if (!(o instanceof ColorPoint) return false;
  ColorPoint cp = (ColorPoint)o;
  return super.equals(o) && cp.color == color;
}

Point p = new Point(1, 2);
ColorPoint cp = new ColorPoint(1, 2, Color.RED);
```


- !! (SURPRISE) !!
- p.equals(cp) => true
- cp.equals(p) => false
- violates symmetry!

Object equality
Problems

// Try 2
```
public boolean equals(Object o) {
  if (!(o instanceof Point)) return false;
  // If o is a normal Point, do a color-less comparison
  if (!(o instanceof ColorPoint)) return o.equals(this);
  // o is a ColorPoint; do a full comparison
  ColorPoint cp = (ColorPoint)o;
    return super.equals(o) && cp.color == color;
}
```

- This approach does provide symmetry
- But ...

Object equality
Problems

// Try 2
```
public boolean equals(Object o) {
  if (!(o instanceof Point)) return false;
  // If o is a normal Point, do a color-less comparison
  if (!(o instanceof ColorPoint)) return o.equals(this);
  // o is a ColorPoint; do a full comparison
  ColorPoint cp = (ColorPoint)o;
    return super.equals(o) && cp.color == color;
}
```

- This approach does provide symmetry
- But ...


```
ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point(1, 2);
ColorPoint p3 = new ColorPoint(1, 2, Color.BLUE);
```

- p1.equals(p2) and p2.equals(p3) => true,
- p1.equals(p3) => false
- Transitivity violation

Object equality
Problems

- Solution ??
- Fundamental problem of equivalence relations in object-oriented languages.
- **No alternative – better say:** no simple way to preserve equals when
  - **extending an instantiable class and**
  - **adding an extra attribute**
- How to overcome the problem?
  - Use the guideline "Favor composition over inheritance"
  - ColorPoint defines a Point object as an instance variable

```
// Solution: adding an attribute without
// violating the equals contract

public class ColorPoint {

  private Point point;
  private Color color;

  public ColorPoint(int x, int y, Color color) {
    point = new Point(x, y);
    this.color = color;
  }

  public Point getPoint() { return point; }

  public boolean equals(Object o) {
    if (!(o instanceof ColorPoint))
        return false;
    ColorPoint cp = (ColorPoint)o;
    return cp.getPoint().equals(point) &&

      cp.color.equals(color);
  }
  // … The rest of class resources
```


Object equality
Problems

- Examples of Java libraries
  - java.sql.Timestamp subclasses java.util.Date
    - add nanoseconds field.
    - equals implementation for Timestamp violate symmetry
    - problems if Timestamp and Date objects are used in the same collection
- Note
  - Adding an attribute to a subclass of an *abstract* class
    - no equals contract violation because abstract classes could not be instantiated
    - Example:
    - Shape, Circle, Rectangle classes

Object equality
Final Recommendations

- Override hashCode when override equals
- Avoid equals method using unreliable resources
  - Example
  - java.net.URL's equals method
  - Relies on the IP addresses of the hosts in URLs being compared
  - Translating a host name to an IP address can require network access, and it isn't guaranteed to yield the same results over time
  - This can cause the URL equals method to violate the equals contract, and it has caused problems in practice
- Don't substitute Object type in equals method declaration

public boolean equals(NotObjectClass o) { ...}

  - Is not overriding Object equals
  - Strong typed equals
    - Acceptable to be provided in addition to the normal equals

UTCN - Programming Techniques 29

hash Code

- hashCode
  - Returns a hash code value for the object of invocation
  - Used by the hash-based collections such as HashMap, HashSet, and Hashtable
- Main rule
  - **Equal objects must have equal hash codes**
  - Method hashCode() should be overridden for all classes that overrides equals
- Main rule violation
  - Violation of the general contract for Object.hashCode
  - Prevents proper operation of the class when using hash-based collections
- hashCode contract (from JDK):
  - When hashCode is invoked on the same object during the execution of a Java application => same integer result
  - hashCode invocation on two equal objects (according to equals method) => same integer result
- Case of two unequal objects and hashCode
  - if !o1.equals(o2) => hashCode(o1) might be equal to hashCode(o2)
  - Improve hashtable performances when

!o1.equals(o2) =>
hashCode(o1) != hashCode(o2)

UTCN - Programming Techniques 30

hash Code

- Calculate a hash code for each significant field
  - the significant fields
    - those that are compared in the equals method
  - primitive type field => convert to integer
  - reference type field => call hashCode for that field
- Combine the hash codes of all significant fields

- Template for calculating the hashCode

```
public int hashCode() {
  int hash = 0; // cumulative
  int c;  // field hash code
  // for each field …
  // … compute and combine the
    hash code
  return hash;
}
```


hash Code

- Examples of combining the hash codes of individual fields
  - Bitwise or

    hash = hash << n | c

  n is an arbitrary integer constant
  - Addition

    hash = hash * p + c

      p is a prime number

Example

```
public int hashCode() {
 int sum = 0;
 DLnode node = head;
 while(node != null) {
   if(node.element != null) {
     sum <<= 8;
     sum |= node.element.hashCode()
              & 0xFF;
   }
   node = node.next;
 }
 return sum;
}
```

Cloning objects

- Method clone() is defined in the class Object

protected Object clone()

- Returns a copy of (this) object instance
  - Default implementation: Field by field copy or shallow copy
- Similar to C++ copy constructor
- Interface Clonable
  - Marker interface
- A class that inherits the clone() method from Object can have objects cloned only if it implements Cloneable
  - If the class is not implementing Cloneable, throws CloneNotSupportedException
- Note. A class may override clone() with its own implementation that ignores the presence or absence of interface Clonable

Cloning objects

clone() contract

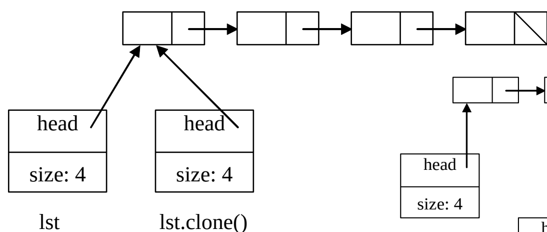
- Creates and returns a copy of this object
- “copy” meaning
- General intent
  - for any object x, the following expressions are true
    - x.clone() != x
    - x.clone().getClass() == x.getClass()
    - x.clone().equals(x)

- Cloning pattern for a class C

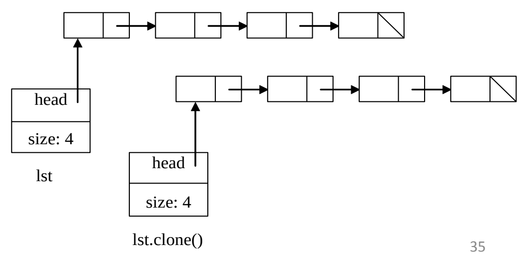
```
public Object clone() throws
 CloneNotSupportedException {
  C clone = (C)super.clone();
  // … do cloning of reference
  // type fields for deep copy
  return clone;
}
```

Cloning objects
Shallow and Deep copy

- Shallow copy
  - Field by field copy
  - No matter the field is primitive type or reference type
    - ok for primitive type fields
    - for reference types - the referenced objects are not cloned



- Deep copy
  - Objects referenced by reference types are also cloned



Cloning objects

Example - Shallow copy
- Point supporting clone
- Class defines only primitive data types
- shallow copy defined in the Object class is OK

```
public class Point implements Cloneable {
   private double x, y; // coordinates
   public Object clone() throws

      CloneNotSupportedException {
      return super.clone();
   }
   //… other class resources
}

// Client code
Point p1 = new Point(2.1, 3.3);
Point p2 = (Point)p1.clone();
```

Example – Deep copy of a Doubly Linked List

```
public class DLlist implements List, Cloneable {
  // attributes
  protected DLnode head, tail;
  protected int howMany;

  public Object clone() throws
                CloneNotSupportedException {
    DLlist list = (DLlist)super.clone();
    list.head = list.tail = null;
    list.howMany = 0;
    for(DLnode node =head; node != null; node =
                                    node.next) {
        if(node.element != null) {
          list.addLast(node.element);
        }
    }
    return list;
  }
}
```

Cloning objects

- Shallow copies are acceptable for objects that contain references to immutable objects and/or to primitives,
- More complex object structures usually require deep copies
- When a deep copy is needed, it's your responsibility to implement the functionality

String representation of Objects toString

- Method toString() - returns a String representation of an object
- Useful in testing and debugging
- The result should include all object fields
- How to use it
  - Explicitly (in debug for example)
  - Implicitly whenever an object reference is specified as part of a string expression

Note. If a class C fails to override toString the default implementation in Object simply displays the name of the object's class and the object's hash code value, separated by the at (@) symbol **C@28ccdaf5**

Loose Coupling

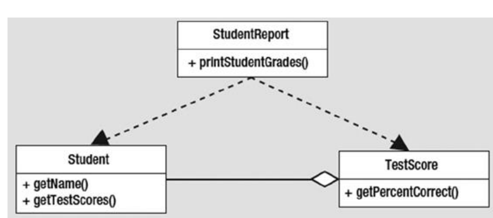
- Degree to which classes depend upon one another
  - Tightly coupled - Two classes that are highly dependent
- Coupling is inevitable
  - Classes must maintain references to one another and
  - Perform method calls
- Guideline: When implementing for class for reuse limit its dependencies on other classes as much as possible
  - It's not obvious how to do this
  - You cannot simply eliminate the interaction between classes
  - Solutions
    - Create a pure abstraction that handles the interaction between two classes or
    - Shift the responsibility for the interaction to an existing class that you don't intend to make reusable

Loose Coupling

```
import java.util.List;
public class Student {
  private List<TestScore> testScores;
  private String name;
  public Student(List<TestScore> scores, String name) {
    this.testScores = scores;
    this.name = name;
  }
  public String getName() { return name; }
  public List<TestScore> getTestScores() {
      return testScores;
  }
}

public class TestScore {
  private int percentCorrect;
  public TestScore(int percent) {
    this.percentCorrect = percent;
  }
  public int getPercentCorrect() {
    return percentCorrect;
  }
}
```

```
import java.util.List;
public class StudentReport {
  public void printStudentGrades(Student[]   students) {
    List<TestScore> testScores;
    int total;
    for (Student student : students) {
      testScores = student.getTestScores();
      total = 0;
      for (TestScore testScore : testScores) {
        total += testScore.getPercentCorrect();
      }
      System.out.println("Final grade for " +
                        student.getName() + " is " +
                        total / testScores.size());
    }
  }
}
```

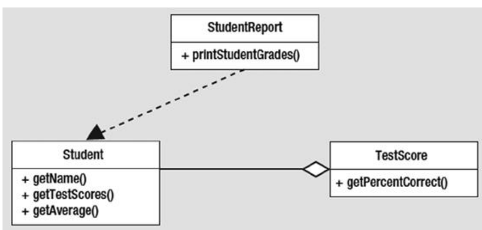


Source: Brett, Spell

Loose Coupling

The problems

- **StudentReport** - is coupled both to Student and to TestScore (because TestScore contains the information needed to calculate the averages)
- StudentReport - unnecessary **tight coupling** (its dependency upon TestScore)
- StudentReport - **weak cohesion** due to performing two functions:
  - printing a report and
  - calculating each student's average.




```
import java.util.List;
public class Student {
  // ... previos code
  public int getAverage() {
    int total = 0;
    for (TestScore testScore : testScores) {
      total += testScore.getPercentCorrect();
    }
    return total / testScores.size();
  }
}

public class StudentReport {
  public void printStudentGrades (Student[] students) {
    for (Student student : students) {
        System.out.println("Final grade for " +
                student.getName() +
                " is " + student.getAverage());
    }
  }
}
```


Quality Interfaces

- Strong Cohesion
- Completeness
- Convenience
- Clarity
- Consistency

Quality Interfaces
Strong Cohesion

- Highly cohesive interface
  - all methods are closely related and are complete
- An interface isn't cohesive if
  - Contains methods that perform unrelated functions or
  - Some set of closely related functions is split across that class and other classes
  - Too much functionality is added to a single class
  - **Good rule of thumb**: keep the responsibilities of a class limited enough that they can be outlined with a brief description
  - If a class has unrelated responsibilities, split it up into two classes
- Condition for a public interface of a class to be cohesive:
  - The class features should be related to a single abstraction
- Example
  - The following code runs ok but it features strong coupling and low cohesion

Quality Interfaces
Completeness

- Completeness
- A class interface should be complete. It should support all operations that are a part of the abstraction that the class represents.

Quality Interfaces
Convenience

- The interfaces should provide convenient (and easy ways) ways to accomplish common tasks
- Example
- Common task of reading input from System.in
- Before Java 5.0
  - System in has to be wrapped into an InputStreamReader and then into a BufferedReader (inconvenient)
- After Java 5.0
  - Scanner class solved this problem in a more convenient way

Quality Interfaces
Clarity

- The interface of a class should be clear to programmers, without generating confusion
  - Confused programmers write buggy code

Example - Adding / Removing list items while iteration

- Adding (intuitive way, see cursor in the comment)

```
ListIterator<String> iterator = list.listIterator( ); // I ABC
iterator.next(); // A I BC
iterator.add("X" ); // AX I BC
```

- Removing (non-intuitive if word processing analogy, i.e. Backspace key is used) - The code below is illegal

```
iterator.remove(); // A I BC
iterator.remove(); // I BC
```

- The rule reads
  - Removes from the list the last element that was returned by next or previous**. This call can only be made once per call to next or previous**. It can be made only if add has not been called after the last call to next or previous

Quality Interfaces
Consistency

- The operations in a class should be consistent with each other with respect to names, parameters and return values, and behavior

Example

1. Constructor of GegorianCalendar in java.util

- GregorianCalendar(int year, int month, int dayOfMonth)
  - month: 0 ..11
  - dayOfMonth: 1 .. 31

2. String related equals and regionMatching

s.**equals**(t) ;
s.**equalsIgnoreCase**(t) ;

boolean **regionMatches** (int toffset, String other, int ooffset, int len)
boolean **regionMatches** (boolean **ignoreCase**, int toffset, String other, int ooffset , int len)