



Chapter 5: Reusability-Oriented Software Construction Approaches

5.2 Construction for Reuse

Xu Hanchuan

xhc@hit.edu.cn

April 8, 2018

Outline

Designing reusable classes

- Inheritance and overriding
- Behavioral subtyping and overloading
- Parametric polymorphism and generic programming
- Composition and delegation

Designing system-level reusable libraries and frameworks

- API and Library
- Framework
- Java Collections Framework (an example)





1 Designing reusable classes





(1) Inheritance and overriding see 3.4





(2) Behavioral subtyping and overloading see 3.4





(a) Three Types of Polymorphism

Three Types of Polymorphism (多态)

- A **polymorphic type** is one whose operations can be applied to values of some other types.(多态类型,其操作可以用于其他类型的值)
 - Ad hoc polymorphism: when a function denotes different and potentially heterogeneous(异构的) implementations depending on a limited range of individually specified types and combinations. Ad hoc polymorphism is supported in many languages using function overloading. 一个函数可以有多个同名的实现
 - Parametric polymorphism(参数多态性): when code is written without mention of any specific type and thus can be used transparently with any number of new types. In the object-oriented programming community, this is often known as generics or generic programming. 一个类型名字可以代表多个类型
 - Subtyping (also called subtype polymorphism or inclusion polymorphism): when a name denotes instances of many different classes related by some common superclass. (We have already discussed it in previous section) 一个变量名字可以代表多个类的实例





(b) Ad hoc polymorphism and Overloading see 3.4





(c) Parametric polymorphism and generic programming see 3.4





(d) Subtyping Polymorphism see 3.4





(e) Behavioral subtyping and Liskov Substitution Principle (LSP)

Behavioral subtyping

Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.

– Barbara Liskov

- Compiler-enforced rules in Java:
 - Subtypes can add, but not remove methods
 - Concrete class must implement all undefined methods
 - Overriding method must return same type or subtype
 - Overriding method must accept the same parameter types
 - Overriding method may not throw additional exceptions
- Also applies to specified behavior:
 - Same or stronger invariants
 - Same or stronger postconditions for all methods
 - Same or weaker preconditions for all methods

Liskov
Substitution
Principle
(LSP)

- Subclass fulfills the same invariants (and additional ones)
- Overridden method has the same pre and postconditions

```
abstract class Vehicle {
                                            class Car extends Vehicle {
         int speed, limit;
                                                      int fuel;
                                                      boolean engineOn;
                                                      //@ invariant speed < limit;
         //@ invariant speed < limit;
                                                      //@ invariant fuel >= 0;
                                                      //@ requires fuel > 0 && !engineOn;
                                                      //@ ensures engineOn;
                                                      void start() { ... }
                                                      void accelerate() { ... }
         //@ requires speed != 0;
                                                      //@ requires speed != 0;
         //@ ensures speed < \old(speed)
                                                      //@ ensures speed < \old(speed)</pre>
         void brake();
                                                      void brake() { ... }
```

- Subclass fulfills the same invariants (and additional ones)
- Overridden method start has weaker precondition
- Overridden method brake has stronger postcondition

```
class Car extends Vehicle {
                                                    class Hybrid extends Car {
          int fuel;
                                                               int charge;
                                                               //@ invariant charge >= 0;
          boolean engineOn;
          //@ invariant fuel >= 0;
                                                               //@ requires (charge > 0 || fuel > 0)
                                                                              && !engineOn;
          //@ requires fuel > 0 && !engineOn;
          //@ ensures engineOn;
                                                               //@ ensures engineOn;
                                                               void start() { ... }
          void start() { ... }
          void accelerate() { ... }
                                                               void accelerate() { ... }
          //@ requires speed != 0;
                                                               //@ requires speed != 0;
          //@ ensures speed < old(speed)
                                                               //@ ensures speed < \old(speed)
          void brake() { ... }
                                                               //@ ensures charge > \old(charge)
                                                               void brake() { ... }
```

• How about these two classes? Is LSP satisfied?

```
class Rectangle {
                                         class Square extends Rectangle {
                                                  Square(int w) {
         int h, w;
                                                       super(w, w);
         Rectangle(int h, int w) {
              this.h=h; this.w=w;
         //methods
                                         class Square extends Rectangle {
class Rectangle {
         //@ invariant h>0 && w>0;
                                                  //@ invariant h>0 && w>0;
                                                  //@ invariant h==w;
         int h, w;
                                                  Square(int w) {
                                                      super(w, w);
         Rectangle(int h, int w) {
             this.h=h; this.w=w;
         //methods
```

How about these two classes? Is LSP satisfied?

```
class Rectangle {
         //@ invariant h>0 && w>0;
         int h, w;
         Rectangle(int h, int w) {
              this.h=h; this.w=w;
         //@ requires factor > 0;
         void scale(int factor) {
              w=w*factor;
              h=h*factor;
```

```
class Square extends Rectangle {
    //@ invariant h>0 && w>0;
    //@ invariant h==w;
    Square(int w) {
        super(w, w);
    }
}
```

How about these two classes? Is LSP satisfied?

```
class Rectangle {
                                       class Square extends Rectangle {
                                               //@ invariant h>0 && w>0;
        //@ invariant h>0 && w>0;
        int h, w;
                                               //@ invariant h==w;
                                               Square(int w) {
        Rectangle(int h, int w) {
                                                    super(w, w);
            this.h=h; this.w=w;
        //@ requires factor > 0;
        void scale(int factor) {
                                              class GraphicProgram {
            w=w*factor;
                                                void scaleW(Rectangle r, int factor) {
             h=h*factor;
                                                  r.setWidth(r.getWidth() * factor);
        //@ requires neww > 0;
        void setWidth(int neww) {
            w=neww;
                                       Invalidates stronger invariant
                                                (w==h) in subclass
```





(3) Delegation and Composition

A Sorting example

Version A:

```
static void sort(int[] list, boolean ascending) {
    ...
    boolean mustSwap;
    if (ascending) {
        mustSwap = list[i] < list[j];
    } else {
        mustSwap = list[i] > list[j];
    }
    ...
}
```

Version B:

```
interface Comparator {
  boolean compare(int i, int j);
}
final Comparator ASCENDING = (i, j) -> i < j;
final Comparator DESCENDING = (i, j) -> i > j;

static void sort(int[] list, Comparator cmp) {
    ...
   boolean mustSwap =
      cmp.compare(list[i], list[j]);
   ...
}
```

Delegation 委托

- Delegation is simply when one object relies on another object for some subset of its functionality (one entity passing something to another entity)
 - e.g. here, the Sorter is delegating functionality to some Comparator
- Judicious delegation enables code reuse
 - Sorter can be reused with arbitrary sort orders
 - Comparators can be reused with arbitrary client code that needs to compare integers
- Explicit delegation: passing the sending object to the receiving object
- Implicit delegation: by the member lookup rules of the language
- Delegation can be described as a low level mechanism for sharing code and data between entities.

A simple Delegation example

```
class RealPrinter { // the "receiver"
   void print() {
       System. out. println("Hello world!");
class Printer { // the "sender"
   RealPrinter p = new RealPrinter(); // create the receiver
   void print() {
       p. print(): // calls the receiver
public class Main {
   public static void main(String[] arguments) {
              // to the outside world it looks like Printer actually prints.
       Printer printer = new Printer();
       printer.print():
```

A simple Delegation example

```
interface I
   void f():
   void g();
class A implements I
   public void f() { System.out.println("A: doing f()"); }
   public void g() { System.out.println("A: doing g()"); }
class B implements I
   public void f() { System.out.println("B: doing f()"); }
   public void g() { System.out.println("B: doing g()"); }
// changing the implementing object in run-time (normally done in compile time)
class C implements I {
   I i = null:
   // forwarding
   public C(I i) { setI(i); }
   public void f() { i. f(); }
   public void g() { i.g(); }
   // normal attributes
   public void setI(I i) { this. i = i; }
public class Main {
                                                                     Switch to
   public static void main(String[] arguments) {
       C c = new C(new A());
                                                                      another
       c.f(); // output: A: doing f()
       c.g(); // output: A: doing g()
                                                                    forwarder
       c. setI (new B());
       c. f(); // output: B: doing f()
       c.g(); // output: B: doing g()
```

Delegation

- The delegation pattern is a software design pattern for implementing delegation, though this term is also used loosely for consultation or forwarding.
- Delegation is dependent upon dynamic binding, as it requires that a given method call can invoke different segments of code at runtime.

Process

- The Receiver object delegates operations to the Delegate object
- The Receiver object makes sure, that the Client does not misuse the Delegate object.



Using delegation to extend functionality

Consider java.util.List

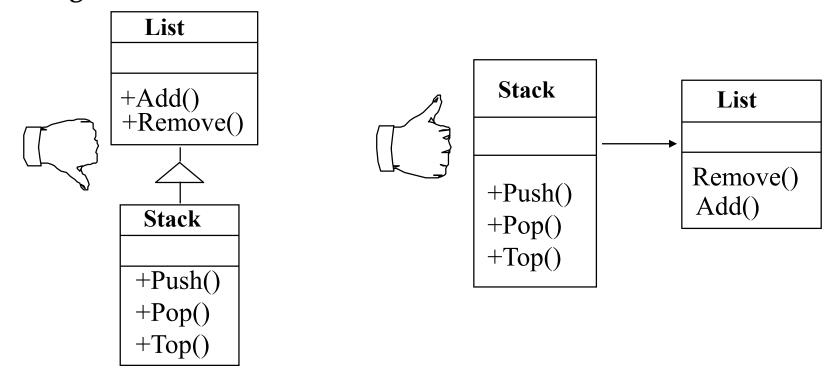
```
public interface List<E> {
  public boolean add(E e);
  public E         remove(int index);
  public void      clear();
  ...
}
```

- Suppose we want a list that logs its operations to the console...
 - The LoggingList is composed of a List, and delegates (the non-logging) functionality to that List.

```
public class LoggingList<E> implements List<E> {
   private final List<E> list;
   public LoggingList<E>(List<E> list) { this.list = list; }
   public boolean add(E e) {
        System.out.println("Adding " + e);
        return list.add(e);
   }
   public E remove(int index) {
        System.out.println("Removing at " + index);
        return list.remove(index);
   }
}
```

Delegation vs. Inheritance

- Inheritance: Extending a Base class by a new operation or overwriting an operation.
- Delegation: Catching an operation and sending it to another object.
- Many design patterns use a combination of inheritance and delegation.



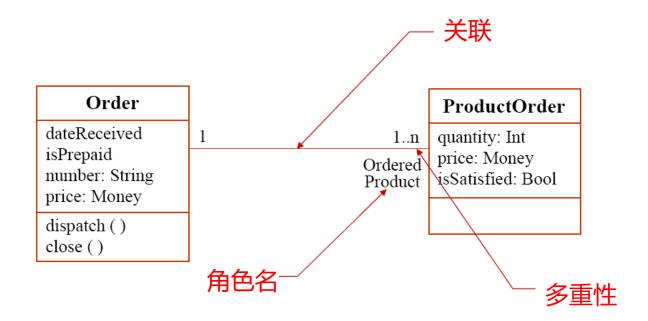
Delegation vs. Inheritance

```
public class A {
    void foo() { this.bar(); }
    void bar() { System.out.println("A.bar"); }
public class B extends A {
    public B() {}
    void foo() { super.foo(); }
    void bar() { System.out.println("B.bar"); }
    B b = new B();
    b.foo();
```

Calling b.foo() will result in what?

Delegation: association

- Association: a persistent relationship between classes of objects that allows one object instance to cause another to perform an action on its behalf.
 - This relationship is structural, because it specifies that objects of one kind are connected to objects of another and does not represent behavior.

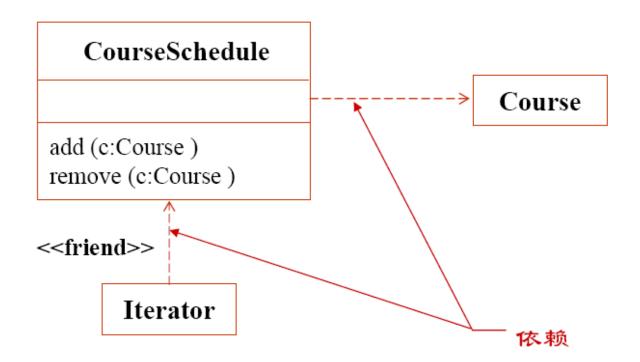


Delegation: association

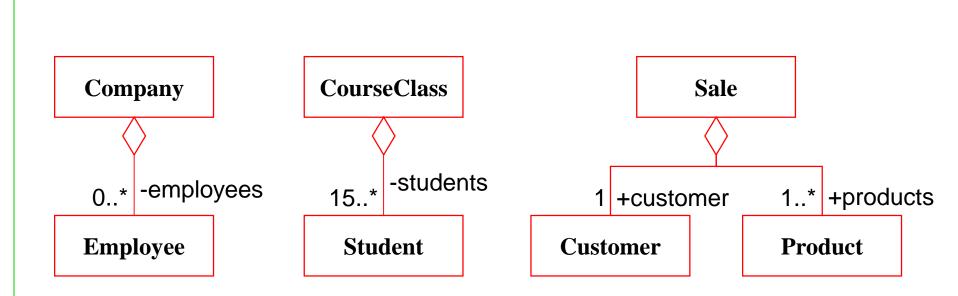
```
- selectedCourses
class Course {}
                               Student
                                                                    Course
class Student {
  private Course [ ] selectedCourses;
class Teacher {
  private Student [ ] students;
                                                       - students
                                Teacher
                                                                    Student
class Student {
                                           - teacher
  private Teacher teacher;
```

Delegation: dependency

 Dependency: a temporary relationship that an object requires other objects (suppliers) for their implementation.

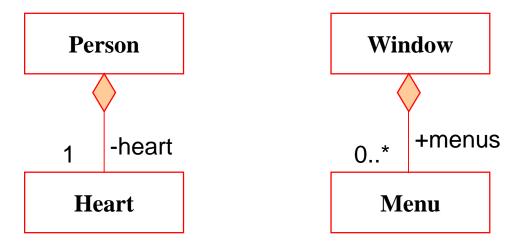


- Composition is a way to combine simple objects or data types into more complex ones.
 - An object of a composite type (e.g. car) "has an" object of a simpler type (e.g. wheel).
 - Composition is implemented such that an object contains another object.
 - A special composition: aggregation
- In composition, when the owning object is destroyed, so are the contained objects. In aggregation, this is not necessarily true.
 - A university owns various departments, and each department has a number of professors. If the university closes, the departments will no longer exist, but the professors in those departments will continue to exist.
 - A University can be seen as a composition of departments, whereas departments have an aggregation of professors. A Professor could work in more than one department, but a department could not be part of more than one university.



A company owns zero or multiple employees; A course's class owns above 15 students; An Sale owns a customer and a set of products;

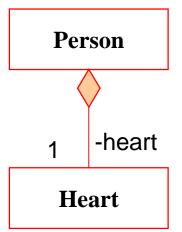
```
定义两个类:
                                                                    CourseClass
class Student {}
                                                                           -students
                                                                     15..*
class CourseClass {
                                                                       Student
                                         使用时的代码:
  private Student[] students;
                                         Student a = new Student ();
  public addStudent (Student s) {
                                         Student b = new Student ();
         studtents.append(s);
                                         Student n = new Student ();
                                         CourseClass SE = new CourseClass();
                                         SE.addStudent (a);
                                         SE.addStudent (b);
                                         SE.addStudent (n);
```



A heart is part of a person; A menu is part of a window;

```
class Heart {}

class Person {
    ...
    private Heart heart = new Heart();
    ...
}
```



Five types of relations between objects

Delegation

- Association
- Dependency
- Composition
- Aggregation
- Inheritance

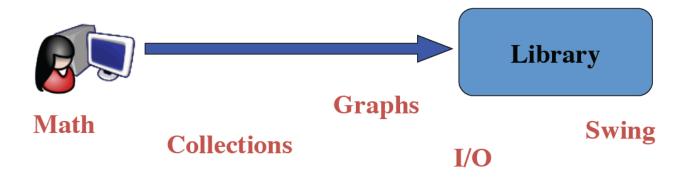




2 Designing system-level reusable libraries and frameworks

Libraries

 Library: A set of classes and methods (APIs) that provide reusable functionality

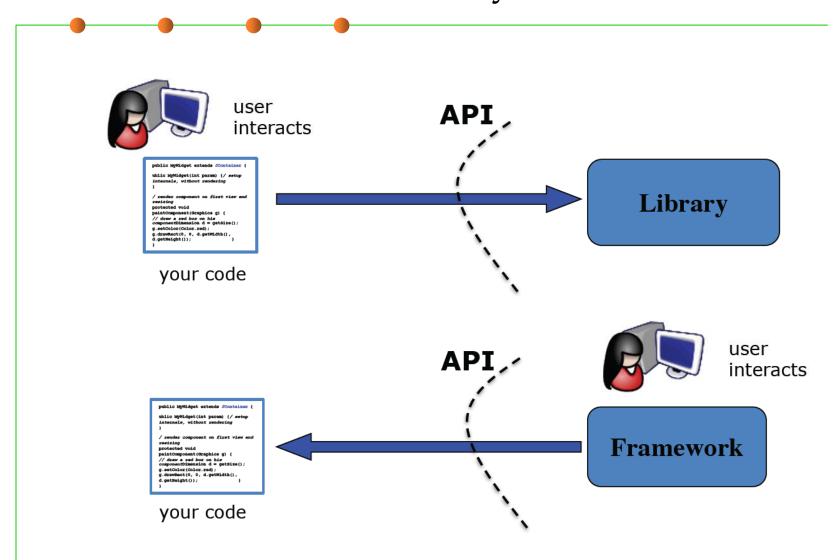


Framework

- Framework: Reusable skeleton code that can be customized into an application
- Framework calls back into client code
 - The Hollywood principle: "Don't call us. We'll call you."

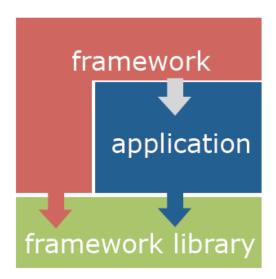


General distinction: Library vs. framework



Libraries and frameworks in practice

- Defines key abstractions and their interfaces
- Defines object interactions & invariants
- Defines flow of control
- Provides architectural guidance
- Provides defaults



More terms

- API: Application Programming Interface, the interface of a library or framework
- Client: The code that uses an API
- Plugin: Client code that customizes a framework
- Extension point: A place where a framework supports extension with a plugin
- Protocol: The expected sequence of interactions between the API and the client
- Callback: A plugin method that the framework will call to access customized functionality
- Lifecycle method: A callback method that gets called in a sequence according to the protocol and the state of the plugin



(1) API design

Why is API design important?

- If you program, you are an API designer, and APIs can be among your greatest assets
 - Good code is modular each module has an API
 - Users invest heavily: acquiring, writing, learning
 - Thinking in terms of APIs improves code quality
 - Successful public APIs capture users
- Can also be among your greatest liabilities
 - Bad API can cause unending stream of support calls
 - Can inhibit ability to move forward
- Public APIs are forever one chance to get it right
 - Once module has users, can't change API at will

Characteristics of a good API

- Easy to learn
- Easy to use, even without documentation
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to evolve
- Appropriate to audience

Sample early API draft

```
// A collection of elements (root of the collection hierarchy)
public interface Collection<E> {
 // Ensures that collection contains o
 boolean add(E o);
 // Removes an instance of o from collection, if present
 boolean remove(Object o);
 // Returns true iff collection contains o
 boolean contains(Object o);
 // Returns number of elements in collection
 int size();
 // Returns true if collection is empty
 boolean isEmpty();
  ... // Remainder omitted
```

(1) API should do one thing and do it well

- Functionality should be easy to explain
 - If it's hard to name, that's generally a bad sign
 - Good names drive development
 - Be amenable to splitting and merging modules
- Good: Font, Set, PrivateKey, Lock, ThreadFactory, TimeUnit, Future<T>
- Bad:
 - DynAnyFactoryOperations
 - BindingIteratorImplBase
 - ENCODING CDR ENCAPS
 - OMGVMCID

(2) API should be as small as possible but no smaller

- API should satisfy its requirements
- When in doubt leave it out
 - Functionality, classes, methods, parameters, etc.
 - You can always add, but you can never remove
- Conceptual weight more important than bulk
- Look for a good power-to-weight ratio 在性能和大小之间找平衡

(3) Implementation should not impact API

- Implementation details in APIs are harmful
 - Confuse users
 - Inhibit freedom to change implementation
- Be aware of what is an implementation detail
 - Do not overspecify the behavior of methods
- For example: do not specify hash functions
 - All tuning parameters are suspect
- Don't let implementation details "leak" into API
 - Serialized forms, exceptions thrown
- Minimize accessibility of everything (information hiding)
 - Make classes, members as private as possible
 - Public classes should have no public fields

(4) Documentation matters

- Document every class, interface, method, constructor, parameter, and exception
 - Class: what an instance represents
 - Method: contract between method and its client
- Preconditions, postconditions, side-effects
 - Parameter: indicate units, form, ownership
- Document thread safety
- If class is mutable, document state space

Recall Chapter 4 for Understandability

Reuse is something that is far easier to say than to do. Doing it requires both good design and very good documentation. Even when we see good design, which is still infrequently, we won't see the components reused without good documentation.

– D. L. Parnas Software Aging, on ICSE 1994

(5) Consider performance consequences

- Bad decisions can limit performance
 - Making type mutable
 - Providing constructor instead of static factory
 - Using implementation type instead of interface
- Do not warp API to gain performance
 - Underlying performance issue will get fixed, but headaches will be with you forever
- Good design usually coincides with good performance
- Performance effects of a bad API decisions can be real and permanent
 - Component.getSize() returns Dimension, but Dimension is mutable, thus each getSize call must allocate Dimension, causing millions of needless object allocations (Dimension类封装了一个构件的高度和宽度)

(6) API must coexist peacefully with platform

Do what is customary

- Obey standard naming conventions
- Avoid obsolete parameter and return types
- Mimic patterns in core APIs and language

Take advantage of API-friendly features

- Generics, varargs, enums, functional interfaces
- Know and avoid API traps and pitfalls
 - Finalizers, public static final arrays, etc.
- Don't transliterate(音译) APIs

(7) Class design

- Minimize mutability: Classes should be immutable unless there's a good reason to do otherwise
 - Advantages: simple, thread-safe, reusable
 - Disadvantage: separate object for each value
 - If mutable, keep state-space small, well-defined.
- Subclass only where it makes sense: Subclassing implies substitutability (LSP)
 - Don't subclass unless an is-a relationship exists. Otherwise, use delegation or composition. 尽量使用委托和组合
 - Don't subclass just to reuse implementation. 不要因为单纯的复用实现而使用继承
 - Inheritance violates encapsulation, and subclasses are sensitive to implementation details of superclass 继承违反了封装原则,且对超类的实 现敏感

(8) Method design

- Don't make the client do anything the module could do
 - Clients generally do via cut-and-paste, which is ugly, annoying, and error-prone.
 import org.w3c.dom.*;
 import invariant invariant invariant.

```
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

/** DOM code to write an XML document to a specified output stream. */
static final void writeDoc(Document doc, OutputStream out)throws IOException{
    try {
        Transformer t = TransformerFactory.newInstance().newTransformer();
        t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, doc.getDoctype().getSystemId());
        t.transform(new DOMSource(doc), new StreamResult(out)); // Does actual writing
    } catch(TransformerException e) {
        throw new AssertionError(e); // Can't happen!
    }
}
```

- APIs should fail fast: report errors as soon as possible. Compile time is best – static typing, generics.
 - At runtime, first bad method invocation is best
 - Method should be failure-atomic(要么正常结束,产生期望结果,要么整个失败,而根本不产生任何的结果)

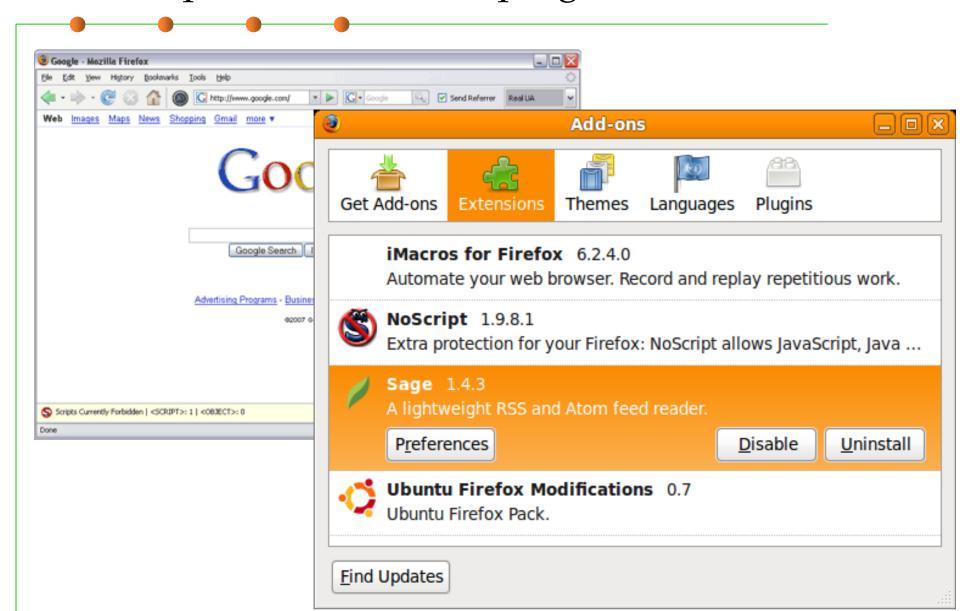
(8) Method design

- Provide programmatic access to all data available in string form.
 Otherwise, clients will parse strings, which is painful for clients
- Overload with care. Often better to use a different name.
- Use appropriate parameter & return types.
 - Favor interface types over classes for input for flexibility, performance
 - Use most specific possible input parameter type, thus moves error from runtime to compile time.
- Avoid long parameter lists. Three or fewer parameters is ideal.
 - What if you have to use many parameters?
- Avoid return values that demand exceptional processing. Return zero-length array or empty collection, not null.



(2) Framework design

An example: web browser plugin



Whitebox and Blackbox frameworks

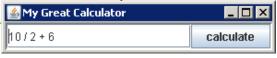
Whitebox frameworks

- Extension via subclassing and overriding methods
- Common design pattern(s): Template Method
- Subclass has main method but gives control to framework

Blackbox frameworks

- Extension via implementing a plugin interface
- Common design pattern(s): Strategy, Observer
- Plugin-loading mechanism loads plugins and gives control to the framework

A calculator example (without a framework)



```
public class Calc extends JFrame {
 private JTextField textField;
 public Calc() {
      JPanel contentPane = new JPanel(new BorderLayout());
      contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
      JButton button = new JButton();
     button.setText("calculate");
      contentPane.add(button, BorderLayout.EAST);
     textField = new JTextField("");
     textField.setText("10 / 2 + 6");
     textField.setPreferredSize(new Dimension(200, 20));
      contentPane.add(textfield, BorderLayout.WEST);
     button.addActionListener(/* calculation code */);
     this.setContentPane(contentPane);
     this.pack();
     this.setLocation(100, 100);
     this.setTitle("My Great Calculator");
```

A simple whitebox framework

```
public abstract class Application extends JFrame {
  protected String getApplicationTitle() { return ""; }
  protected String getButtonText() { return ""; }
  protected String getInitialText() { return ""; }
  protected void buttonClicked() { }
  private JTextField textField;
  public Application() {
      JPanel contentPane = new JPanel(new BorderLayout());
      contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
      JButton button = new JButton();
      button.setText(getButtonText());
      contentPane.add(button, BorderLayout.EAST);
      textField = new JTextField("");
      textField.setText(getInitialText());
      textField.setPreferredSize(new Dimension(200, 20));
      contentPane.add(textField, BorderLayout.WEST);
      button.addActionListener((e) -> { buttonClicked(); });
      this.setContentPane(contentPane);
      this.pack();
      this.setLocation(100, 100);
      this.setTitle(getApplicationTitle());
      . . .
```

Using the whitebox framework

Extension via subclassing and overriding methods Subclass has main method but gives control to framework

```
public class Ping extends Application {
  protected String getApplicationTitle() { return "Ping"; }
  protected String getButtonText() { return "ping"; }
  protected String getInititalText() { return "127.0.0.1"; }
  protected void buttonClicked() { ... }
}
```

An example blackbox framework

```
public interface Plugin {
public class Application extends JFrame {
                                             String getApplicationTitle();
                                             String getButtonText();
   private JTextField textField;
   private Plugin plugin;
                                             String getInititalText();
                                             void buttonClicked();
   public Application() { }
                                             void setApplication(Application app);
   protected void init(Plugin p) {
     p.setApplication(this);
     this.plugin = p;
     JPanel contentPane = new JPanel(new BorderLayout());
     contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
     JButton button = new JButton();
     button.setText(plugin != null ? plugin.getButtonText() : "ok");
     contentPane.add(button, BorderLayout.EAST);
     textField = new JTextField("");
     if (plugin != null)
         textField.setText(plugin.getInititalText());
     textField.setPreferredSize(new Dimension(200, 20));
     contentPane.add(textField, BorderLayout.WEST);
     if (plugin != null)
         button.addActionListener((e) -> { plugin.buttonClicked(); } );
     this.setContentPane(contentPane);
  public String getInput() { return textField.getText(); }
```

Using the blackbox framework

Extension via implementing a plugin interface Plugin-loading mechanism loads plugins and gives control to the framework

Whitebox vs. Blackbox Frameworks

Whitebox frameworks use subclassing

- Allows extension of every nonprivate method
- Need to understand implementation of superclass
- Only one extension at a time
- Compiled together
- Often so-called developer frameworks

Blackbox frameworks use composition

- Allows extension of functionality exposed in interface
- Only need to understand the interface
- Multiple plugins
- Often provides more modularity
- Separate deployment possible (.jar, .dll, ...)
- Often so-called end-user frameworks, platforms

Framework design considerations

- Once designed there is little opportunity for change
- Key decision: Separating common parts from variable parts
 - What problems do you want to solve?
- Possible problems:
 - Too few extension points: Limited to a narrow class of users
 - Too many extension points: Hard to learn, slow
 - Too generic: Little reuse value

"Maximizing reuse minimizes use"

Typical framework design and implementation

- Define your domain
 - Identify potential common parts and variable parts
 - Design and write sample plugins/applications
- Factor out & implement common parts as framework
- Provide plugin interface & callback mechanisms for variable parts
 - Use well-known design principles and patterns where appropriate...
- Get lots of feedback, and iterate

This is usually called "Domain Engineering".

Evolutionary design: Extract Commonalities 获取共性

- Extracting interfaces is a new step in evolutionary design:
 - Abstract classes are discovered from concrete classes
 - Interfaces are distilled from abstract classes
- Start once the architecture is stable
 - Remove non-public methods from class
 - Move default implementations into an abstract class which implements the interface

Running a framework

- Some frameworks are runnable by themselves
 - e.g. Eclipse
- Other frameworks must be extended to be run
 - Swing, JUnit, MapReduce, Servlets
- Methods to load plugins:
 - Client writes main(), creates a plugin and passes it to framework
 - Framework writes main(), client passes name of plugin as a command line argument or environment variable
 - Framework looks in a magic location, and then config files or .jar files are automatically loaded and processed.

Example: An Eclipse plugin

- Eclipse is a popular Java IDE.
- More generally, a framework for tools that facilitate "building, deploying and managing software across the lifecycle."
- Plugin framework based on OSGI standard
- Starting point: Manifest file
 - Plugin name
 - Activator class
 - Meta-data

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: MyEditor Plug-in
Bundle-SymbolicName: MyEditor;
singleton:=true
Bundle-Version: 1.0.0
Bundle-Activator:
 myeditor.Activator
Require-Bundle:
 org.eclipse.ui,
 org.eclipse.core.runtime,
 org.eclipse.jface.text,
 org.eclipse.ui.editors
Bundle-ActivationPolicy: lazy
Bundle-
RequiredExecutionEnvironment:
JavaSE-1.6
```

Example: An Eclipse plugin

plugin.xml

- Main configuration file
- XML format
- Lists extension points

Editor extension

- extension point: org.eclipse.ui.editors
- file extension
- icon used in corner of editor
- class name
- unique id
 - refer to this editor
 - other plugins can extend with new menu items, etc.!

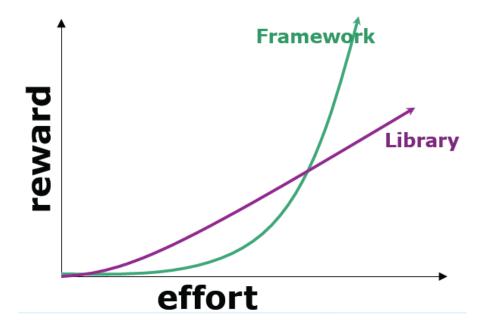
- At last, code!
- XMLEditor.java
 - Inherits TextEditor behavior such as open, close, save, display, select, cut/copy/paste, search/replace, ...
 - Extends with syntax highlighting
 - XMLDocumentProvider partitions into tags and comments
 - XMLConfiguration shows how to color partitions

Example: A JUnit Plugin

```
public class SampleTest {
      private List<String> emptyList;
      @Before ___
      public void setUp() {
          emptyList = new ArrayList<String In JUnit the plugin
                                           mechanism is Java
                                           annotations
     @After <
      public void tearDown() {
          emptyList = null:
     @Test
      public void testEmptyList() {
          assertEquals("Empty list should have 0 elements",
                       0, emptyList.size());
```

Learning a framework

- Documentation
- Tutorials, wizards, and examples
- Other client applications and plugins
- Communities, email lists and forums







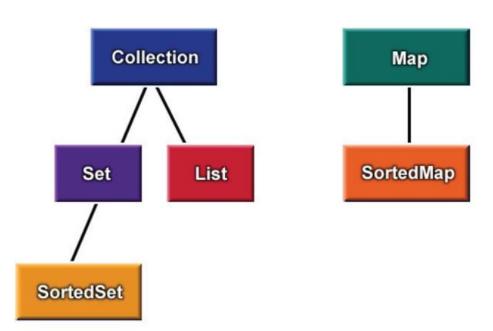
(3) Java Collections Framework

What is a Collection and Collection Framework?

- Collection: an object that groups elements
- Main Uses: data storage and retrieval, and data transmission
- Familiar Examples
 - java.util.Vector
 - java.util.Hashtable
 - Array
- Collection Framework: a Unified Architecture for
 - Interfaces implementation-independence
 - Implementations reusable data structures
 - Algorithms reusable functionality
- Best-known examples
 - C++ Standard Template Library (STL)

Architecture Overview

- Core Collection Interfaces
- General-Purpose Implementations
- Wrapper Implementations
- Abstract Implementations
- Algorithms



Collection Interface

```
public interface Collection<E> {
  int size();
  boolean isEmpty();
  boolean contains(Object element);
 boolean add(E element); // Optional
  boolean remove(Object element); // Optional
  Iterator<E> iterator();
 Object[] toArray();
  T[] toArray(T a[]);
 // Bulk Operations
  boolean containsAll(Collection<?> c);
  boolean addAll(Collection<? Extends E> c); // Optional
 boolean removeAll(Collection<?> c);  // Optional
                                            // Optional
  boolean retainAll(Collection<?> c);
 void clear(); // Optional
```

Iterator Interface

- Replacement for Enumeration interface
 - Adds remove method
 - Improves method names

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); // Optional
}
```

Set Interface

- Adds no methods to Collection!
- Adds stipulation: no duplicate elements
- Mandates equals and hashCode calculation

```
public interface Set<E> extends Collection<E> {
}
```

Set Idioms:

```
Set<Type> s1, s2;
boolean isSubset = s1.containsAll(s2);
Set<Type> union = new HashSet<>(s1);
union = union.addAll(s2);
Set<Type> intersection = new HashSet<>(s1);
intersection.retainAll(s2);
Set<Type> difference = new HashSet<>(s1);
difference.removeAll(s2);
Collection<Type> c;
Collection<Type> noDups = new HashSet<>(c);
```

List Interface: a sequence of objects

```
public interface List<E> extends Collection<E> {
 E get(int index);
 void add(int index, E element);  // Optional
 Object remove(int index); // Optional
 boolean addAll(int index, Collection<? extends E> c);
                                // Optional
 int indexOf(Object o);
 int lastIndexOf(Object o);
 List<E> subList(int from, int to);
 ListIterator<E> listIterator();
 ListIterator<E> listIterator(int index);
```

List Example

Reusable algorithms to swap and randomize:

```
public static <E> void swap(List<E> a, int i, int j) {
    E tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}
private static Random r = new Random();
public static void shuffle(List<?> a) {
    for (int i = a.size(); i > 1; i--)
        swap(a, i - 1, r.nextInt(i));
}
```

List Idioms: List<Type> a, b;

```
a.addAll(b); // Concatenate two lists
a.subList(from, to).clear(); // Range-remove
// Range-extract
List<Type> partView = a.subList(from, to);
List<Type> part = new ArrayList<>(partView);
partView.clear();
```

Map Interface: a Key-Value mapping

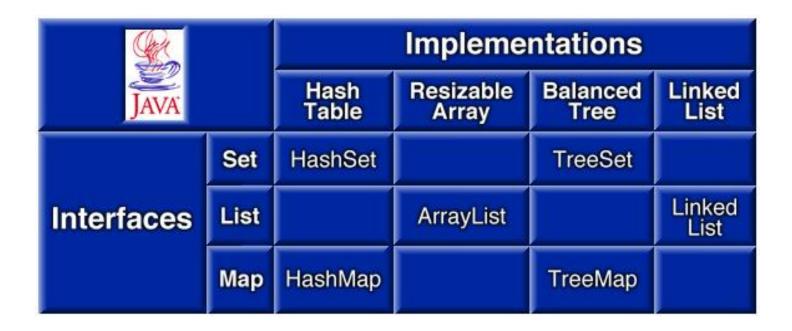
```
public interface Map<K,V> {
  int size();
  boolean isEmpty();
  boolean containsKey(Object key);
  boolean containsValue(Object value);
  Object get(Object key);
  Object put(K key, V value);
                                                    // Optional
  Object remove(Object key);
                                                    // Optional
  void putAll(Map<? Extends K, ? Extends V> t);  // Optional
  void clear();
                                                    // Optional
  // Collection Views
  public Set<K> keySet();
  public Collection<V> values();
  public Set<Map.Entry<K,V>> entrySet();
}
```

Map Idioms

```
// Iterate over all keys in Map m
Map<Key, Val> m;
for (iterator<Key> i = m.keySet().iterator(); i.hasNext(); )
   System.out.println(i.next());
// As of Java 5 (2004)
for (Key k : m.keySet())
   System.out.println(i.next());
// "Map algebra"
Map<Key, Val> a, b;
boolean isSubMap = a.entrySet().containsAll(b.entrySet());
Set<Key> commonKeys =
      new HashSet<>(a.keySet()).retainAll(b.keySet); [sic!]
//Remove keys from a that have mappings in b
a.keySet().removeAll(b.keySet());
```

General Purpose Implementations

Consistent Naming and Behavior



Choosing an Implementation

- Set
 - HashSet -- O(1) access, no order guarantee
 - TreeSet -- O(log n) access, sorted
- Map
 - HashMap -- (See HashSet)
 - TreeMap -- (See TreeSet)
- List
 - ArrayList -- O(1) random access, O(n) insert/remove
 - LinkedList -- O(n) random access, O(1) insert/remove;
 - Use for queues and deques (no longer a good idea!)

Unmodifiable Wrappers

- Anonymous implementations
- Static factory methods
- One for each core interface
- Provide read-only access Immutable!
- We have discussed these wrapper in Chapter 3

Synchronization Wrappers

- Not thread-safe!
- Synchronization Wrappers: a new approach to thread safety
 - Anonymous implementations, one per core interface
 - Static factories take collection of appropriate type
 - Thread-safety assured if all access through wrapper
 - Must manually synchronize iteration
- It was new then; it's old now!
 - Synch wrappers are largely obsolete
 - Made obsolete by concurrent collections

To be discussed in Chapter 10

Synchronization Wrapper Example

```
Set<String> s = Collections.synchronizedSet(new
HashSet<>());
s.add("wombat"); // Thread-safe
synchronized(s) {
  Iterator<String> i = s.iterator(); // In synch block!
  while (i.hasNext())
     System.out.println(i.next());
// In Java 5 (post-2004)
synchronized(s) {
  for (String t : s)
     System.out.println(i.next());
```

Convenience Implementations

- Arrays.asList(E[] a)
 - Allows array to be "viewed" as List
 - Bridge to Collection-based APIs
- EMPTY_SET, EMPTY_LIST, EMPTY_MAP
 - Immutable constants 不支持add操作
- singleton(E o)
 - Immutable set with specified object 返回只包含指定对象的不可变集
- nCopies(E o)
 - Immutable list with n copies of object

Reusable Algorithms

```
static <T extends Comparable<? super T>> void sort(List<T> list);
static int binarySearch(List list, Object key);
static <T extends Comparable<? super T>> T min(Collection<T> coll);
static <T extends Comparable<? super T>> T max(Collection<T> coll);
static <E> void fill(List<E> list, E e);
static <E> void copy(List<E> dest, List<? Extends E> src);
static void reverse(List<?> list);
static void shuffle(List<?> list);
```

Example 1: Sorting lists of comparable elements

```
List<String> strings; // Elements type: String
...
Collections.sort(strings); // Alphabetical order

LinkedList<Date> dates; // Elements type: Date
...
Collections.sort(dates); // Chronological order

// Comparable interface (Infrastructure)
public interface Comparable<E extends Comparable<E>>> {
   int compareTo(Object o);
}
```

Algorithm Example 2: Sorting with a comparator

```
List<String> strings; // Element type: String Collections.sort(strings, Collections.ReverseOrder()); //采用反序比较器
```

```
//自定义比较器: Case-independent alphabetical order
static Comparator<String> cia = new Comparator<>() {
   public int compare(String c1, String c2) {
      return c1.toLowerCase().compareTo(c2.toLowerCase());
   }
};

Collections.sort(strings, cia);

public interface Comparator<T> {
   public int compare(T o1, T o2);
}
```

Compatibility

- Old and new collections interoperate freely
- Upward Compatibility 向上兼容/向前兼容 forward
 - Vector<E> implements List<E>
 - Hashtable<K,V> implements Map<K,V>
 - Arrays.asList(myArray)
- Backward Compatibility 向后兼容/向下兼容 downward
 - myCollection.toArray()
 - new Vector<>(myCollection)
 - new Hashtable<>(myMap)

API Design Guidelines

- Avoid ad hoc collections
 - Input parameter type:
 - Any collection interface (Collection, Map best)
 - Array may sometimes be preferable
 - Output value type:
 - Any collection interface or class
 - Array
- Provide adapters for your legacy collections



Summary



The end

April 8, 2018