Concurrent Systems and Applications

THEODORE HONG
THEODORE.HONG@CL.CAM.AC.UK

Timetable change!

Next Concurrent Systems lecture will be **TUESDAY** at 12.

Next Software Engineering lecture will be Monday at 12 instead.

Outline of the course

- Java review. Objects and classes. Packages, interfaces, nested classes. Design patterns. [3 lectures, Hong]
- Distributed systems. Introduction. TCP and UDP. RPC and RMI. Transactions. Enforcing isolation. Crash recovery and logging. [6 lectures, Hand]
- Advanced Java. Reflection and serialisation. Graphical interfaces. Memory management. Native methods and class loaders. Generic types. [5 lectures, Hong]
- Testing. Software testing strategies. [1 lecture, Blackwell]
- Concurrency. Threads. Mutual exclusion. Deadlock. Condition synchronization. Worked examples. Low-level synchronization. [6 lectures, Harris]

Resources

- Course homepage
  - http://www.cl.cam.ac.uk/teaching/0809/ConcSys/
- Java documentation on the web
  - http://java.sun.com/docs/books/tutorial/
  - http://java.sun.com/javase/6/docs/api/
- Past Tripos questions
  - Concurrent Systems
  - Concurrent Systems and Applications
  - Further Java
  - http://www.cl.cam.ac.uk/teaching/exams/pastpapers/

Reading list

Bruce Eckel, *Thinking in Java* (2006)
http://www.mindview.net/Books/TIJ4/

Erich Gamma et al., *Design Patterns* (1994)

Doug Lea, *Concurrent Programming in Java* (1999)

Jean Bacon, *Concurrent Systems* (2002)

Jean Bacon and Tim Harris, *Operating Systems: Concurrent and Distributed Software Design* (2003)

Glenford Myers et al., *The Art of Software Testing* (2004)

James Gosling et al., *The Java Language Specification* (2005)
http://java.sun.com/docs/books/jls/

Review: Programming with Objects

- Introduction
- Objects and classes
- Packages
- Interfaces
- Nested classes
- Design patterns

Defining an object

```
public class TelephoneEntry {
  public String name;
  public String number;

  public TelephoneEntry(String name, String number) {
    this.name = name;
    this.number = number;
  }

  public String getName() {
    return name;
  }

  public void contact() {
    call(number);
  }
}
```

Defining an object

- What is an object?
- Encapsulation: data + operations
- Class definition, consists of:
  - Field definition
    - this
  - Method definition
    - constructor

Constructors

- Ensure correct initialization of object
- No return value
- If the constructor signature changes, then other classes using old signature will notice
- Overloading constructors
- Default 0-parameter constructor

Creating and using an object

```
TelephoneEntry tel = new TelephoneEntry("John", "76019");
System.out.println(tel.name);

TelephoneEntry tel2 = tel;
tel2.name = "Mary"

System.out.println(tel.name);
```

Is `tel == tel2`?
Is `tel.equals(tel2)`?
Is `tel2.equals(tel)`?

```
tel = null;
System.out.println(tel.name);
```

What happens to `tel2`?

Classes vs. instances

- Can't access non-static fields or methods from static method

Method overloading

```
public class Renderer {
  public void draw(Circle c) {
       ...
  }
  public void draw(Rectangle r) {
       ...
  }
  public void draw(Triangle t) {
       ...
  }
}
```

What about `public int draw(Circle c)` ?

Automatic casting

```
void f(int x, long y) {
    ...
}

void f(long x, int y) {
    ...
}

f(10,10); ?
f(long)10, 10); ?
```

Constructors

```
public class TelephoneEntry {
  public TelephoneEntry(String name, String number) {
    this.name = name;
    this.number = number;

    addToDirectory(this);
  }

  public TelephoneEntry(String name) {
    TelephoneEntry(name, "0");
  }

  ...
}
```

Classes vs. instances

```
public class TelephoneEntry {
  public static final String CAM_PREFIX = "01223";
  private static int entryCount = 0;
  private int ID;

  public static int getEntryCount() {
    return entryCount;
  }

  public TelephoneEntry(String name, String number) {
    this.ID = ++entryCount;
  }
}

int i = TelephoneEntry.getEntryCount();
System.out.println(TelephoneEntry.CAM_PREFIX);
```

What about `System.out.println(TelephoneEntry.getName());` ?

Inheritance

```
public class AddressEntry extends TelephoneEntry {
  public String addr;

  public AddressEntry(String name, String number, String addr)
  {
    super(name, number);
    this.addr = addr;
  }

  public String getAddr() { ... }

  public void contact() {  // override
    if (!call(number))
      write(addr);
  }
}
```

Inheritance

- Inheritance typically models 'is a' relationship, e.g.
  - between Bicycle and a more general PersonnelTransport
  - between SpaceElevator and a more general SatelliteLauncher
- defines more specialised sub-class in terms of existing super-class.
  - Reuse existing fields
  - Add new fields
  - Add new methods
  - Override existing methods
- Anything acting on the superclass can act on the subclass
- Implicitly inherit from java.lang.Object, and calls the Object() constructor
- clone(), equals(), etc.

Inheritance vs. composition

```
public class TransportationVehicle {}

public class Car extends TransportationVehicle {
  public Wheel[] wheels;
  public Engine engine;
}

public class Plane extends TransportationVehicle {
  public Wing[] wings;
  public Tail tail;
}
```

`Car` is a `TransportationVehicle`
`Car` has an `Engine`

Inheritance and references

```
class A { ... }
class B extends A { ... }

A objectA = new A();
B objectB = new B();

A refToA;
B refToB;

refToA = objectA;
refToA = objectB;
```

Do these assignments work?

```
refToB = refToA;
refToB = (B) refToA;
refToB = (B) objectA;
```

Object references

- Objects instantiated by new
  - (constructor implicitly called)
- Objects manipulated through references
  - either a particular instance, or null
- Two references can refer to the same object
- Different types of object equality
- Garbage collection

Arrays and inheritance

What is the relationship between the types `A[]` and `B[]`?

```
A[] arrayOfA = new A[10];
B[] arrayOfB = new B[10];
```

Do these assignments work?

```
arrayOfA[5] = new A();
arrayOfA[5] = new B();

arrayOfB[6] = new A();
arrayOfB[6] = new B();

A[] temp;
temp = arrayOfB;
temp = (A[]) arrayOfB;
```

What can you put in temp?

Methods and inheritance

```
class A {
  A f() { return new A(); }
}

class B extends A {
  A f() {
    System.out.println("override");  return new A();
  }
  A f(int x) {
    System.out.println("overload");  return new A();
  }
  B f() {
    System.out.println("covariant override"); return new B(); }
  }
}
```

Overloading

- Distinct parameter lists can distinguish overloaded method calls
- Distinct return types not enough

Methods and inheritance (2)

```
class A {
  void f() { System.out.println("superclass"); }
}
class B extends A {
  void f() { System.out.println("subclass"); }
  void f2() { super.f(); }
}
```

What happens here?

```
B objectB = new B();
objectB.f();
((A) objectB).f();
objectB.f2();
```

Methods and inheritance (3)

```
public class Renderer {
   public void draw(Shape s) {
         s.draw();
   }
}

public class Circle extends Shape {
  public void draw() { ... }
}

public class Rectangle extends Shape {
  public void draw() { ... }
}

public class Triangle extends Shape {
  public void draw() { ... }
}
```

The dual of polymorphism.

Thought questions

- super.super is not valid syntax in Java even though it might appear to provide a means to access the super-class of a class' super-class. Why might the designers have disallowed this?
- Why can't a class inherit from more than one superclass? What would be the advantages and disadvantages of allowing this?
- Some languages support dynamic inheritance, where a class can change which other class it extends at runtime. Why might this be useful? How might this cause confusion?

Thought questions

- If B changes which class it inherits from (or removes it altogether), C won't find out
- Multiple inheritance can be useful, but diamond problem – ambiguity. More subtly, how many copies of common ancestor exist?
- Dynamic inheritance can help solve multiple inheritance problem. Also, combining hierarchies. Obvious problems.

Packages

Java groups classes into packages.

```
package org.apache.commons.codec;
class Decoder { ... }
class Encoder { ... }

decode(new org.apache.commons.codec.Decoder(), s);

import org.apache.commons.codec.Decoder;
import org.apache.commons.codec.*;
import org.apache.commons.*;  ?
```

Packages

- Classes within a package are typically written by co-operating programmers and are expected to be used together.
- fully qualified name = package + class name
- Don’t have to create the package in any way; just quote the name in any package statement.
- Some compilers create subdirectories in the file system, nesting one directory level for each full stop in the package’s fully-qualified name.
- Packages don't inherit

Access modifiers

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	
`no modifier`	Y	Y		
`private`	Y			

Access modifiers (2)

```
public class A {
  protected int field1;
}
```

In another package...

```
public class B extends A {
  public void method2(B b_ref, A a_ref) {
    System.out.println(field1);
    System.out.println(b_ref.field1);
    System.out.println(a_ref.field1);   ?
  }
}
```

final

```
class FinalField {
  final String A = "Initial value";
  final String B;

  public static final int BLUE = 1;
  public static final int WHITE = 2;
  public static final int RED = 3;

  FinalField() {
    B = "Initial value";
  }

  final void doCriticalTask() { ... }
}
```

abstract

```
public class A {
  abstract int methodName();
}

public class B extends A {
  int methodName() {
    return 42;
  }
}
```

References to abstract classes

```
public abstract class A {
  int x = 20;
  abstract void method1(A a);
}

public class B extends A {
  void method1(A a){
    System.out.println("The A's x is "+a.x);
  }

  void method2(A a) {
    a.method1(a);
    ((B) a).method1(a);
  }
}

public static void main(String [] args) {
    B objectB = new B();
    objectB.method1(new A()); // ?
    objectB.method1(new B()); // ?
    objectB.method2(new B()); // ?
}
```

Combining common functionality

```
public abstract AbstractList {
  public abstract Object get(int index);

  public List subList(int fromIndex, int toIndex) {
    ...
  }
}

class java.util.AbstractMap
class java.util.AbstractList
```

static

- Can be applied to any method or field definition (also nested classes)
- The field/method is associated with the class as a whole rather than with any particular object.
- There is only one value for the whole class, rather than a separate value for each object.
- Similarly, static methods are not associated with a current object—unqualified instance field names and the this keyword cannot be used.
- Static methods called by explicitly naming the class within which the method is defined. The named class is searched, then its super-class, etc. Otherwise the search begins from the class in which the method call is made.

Other modifiers

- strictfp – method/class implemented using IEEE 754/854 floating point arithmetic; identical results on all hardware
- synchronized - only one thread can access the class/method at a time.
- volatile – variable re-read from memory each time, caching not permitted between threads
- transient – in Serialization API, not sent over the network when classes are copied from machine to machine
- native - implemented in native code

Interfaces

```
public interface Set {
  boolean add(E e);
  boolean remove(Object o);
}

public interface SortedSet {
  E first();
}

public class HashSet implements Set
public class ArraySet implements Set

public class ArrayList implements List
public class LinkedList implements List
```

Interfaces

- Groups of classes that provide different
- implementations of the same kind of functionality.
- · e.g. the collection classes in java.util—HashSet and
- ArraySet provide set operations; ArrayList and
- LinkedList provide list-based operations.
- In that example there are some operations available on all
- collections, further operations on all sets, and a third set of
- operations on the HashSet class itself.
- Inheritance and abstract classes can be used to move
- common functionality into super-classes such as
- Collection and Set.
- · Each class can only has a single super-class (in Java),
- so should HashSet extend a class representing the
- hashtable aspects of its behaviour (capacity, load factor), or a class
- representing the set-like operations available on it?
- More generally, it is often desireable to separate the
- definition of a standard programming interface (e.g.
- set-like operations) from their implementation using an
- actual data structure (e.g. a hash table).

Interfaces

- Each Java class may extend only a single super-class, but it can implement a number of interfaces.
- An interface definition just declares method signatures and static final fields (constants).
- An ordinary interface may have public or default access.
- javaAll methods and fields are implicitly public.
- An interface may extend one or more super-interfaces.
- A class that implements an interface must either: supply definitions for each of the declared methods; or be declared an abstract class.

Nested classes

A nested class/interface is one whose definition appears inside another class or interface.

There are four cases:

- inner classes – enclosed class is an ordinary (non-static) class
- static nested classes - enclosed definition is declared static;
- nested interfaces - interface is declared within an enclosing class or interface
- anonymous inner classes.

Nested classes

In general nested classes are used:

- · (i) for programming convenience to put classes nearby
- · (ii) logical grouping of classes
- · (iii) to provide one class with access to private
- members or local variables from its enclosing class.
- Helper classes

Nested classes

- An inner class definition associates each instance of the enclosed class with an instance of the enclosing class, e.g.

```
class Bus {
Engine e;

  class Wheel {
   ...
   }
 }
```

Each instance of Wheel is associated with an enclosing instance of Bus. e.g. Wheel methods can access the field without qualification or access the enclosing Bus as Bus.this.

Can Bus find out what Wheels are associated with it?

- static nested class - not associated with any instance of an enclosing class.
- nested interfaces – implicitly static

Anonymous inner class

A short-hand way of defining inner classes.

```
class A {
  void method1() {
  Object ref = new Object() {
      void method2() {};
    };
  }
}
```

An anonymous inner class may be defined using an interface name rather than a class name—providing inline implementations of all the methods.

```
class A {
  void method1() {
    Ifc i = new Ifc() {
        public void interfaceMethod() {
      };
    };
  }
}
```

Event adapter

```
//An example of using an anonymous inner class.
public class MyClass extends Applet {
    ...
    someObject.addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
          ...//Event listener implementation goes here...
            }
      });
    ...
    }
}
```

MouseAdapter is an empty MouseListener implementation.

Example: Closures

An enthusiast for programming with closures proposes a new language extending Java so that the following method definition would be valid:

```
Closure myCounter(int start) {
  int counter = start;
  return {
    System.out.println(counter++);
  }
}

interface Closure {
  void apply();
}
```

No output is printed when the method is executed – instead it returns an object implementing the Closure interface.

Invoking apply() on the Closure object will cause successive values to be printed.

Closure

```
class Wow {
    public static void main(String[] args) {
        Wow w = new Wow();
        Closure c = w.myCounter(15);
        c.apply();
    }

    Closure myCounter (int start) {
        return new ClosureImplementation(start);
    }

    class ClosureImplementation implements Closure {
        private int counter;

        ClosureImplementation(int start) { counter = start; }
        public void apply () {System.out.println(counter++);}
    }
}
```

Package as inner class

Design patterns

Common idioms frequently emerge in object-oriented programming. Studying these design patterns provides:

- common terminology for describing program organisation and conveying the purposes of inter-related classes; and
- examples of how to structure programs for flexibility and re-use.

Singleton

```
class Singleton {
  static Singleton theInstance = null;
  private Singleton() {...}

  static Singleton getInstance() {
    if (theInstance == null)
      return (theInstance = new Singleton());
    return theInstance;
  }

  void method1() {...}
  void method2() {...}
  void method3() {...}
}
```

Singleton

- Ensures that a class can be instantiated at most once.

- Private constructor ensures external classes cannot
- instantiate the class by calling new.

- A static method creates an instance when first called and
- subsequently returns an object reference to the same
- instance.

- More flexible than a suite of static methods: allows
- sub-classing, e.g. getInstance on ToolKit might return
- MotifToolKit or MacToolKit as appropriate.

- The constraint is enforced (and could subsequently be
- relaxed) in a single place.

Abstract factory

Suppose we have a set of interfaces: Window, ScrollBar, etc., defining components used to build GUIs. There may be several sets of these components—e.g. with different visual appearances.

How does an application get hold of the appropriate instances of classes implementing those interfaces?

One possibility:

```
switch (APPLICATION_MODE) {
  case MACINTOSH: w = new MacWindow(); break;
  case MOTIF : w = new MotifWindow(); break;
  ...
}
```

Abstract factory

Disadvantages

It would be lots of work to add support for a new GUI system.

And we would have to change every application!

A buggy application might try to use a MacWindow with a MotifScrollBar.

Abstract factory

```
public interface GUIFactory {
  Window makeWindow();
  ScrollBar makeScrollBar();
}

public class MacFactory implements GUIFactory {
  Window makeWindow() { return new MacWindow(); }
  ScrollBar makeScrollBar() { return new MacScrollBar(); }
}

public class MotifFactory implements GUIFactory {
  Window makeWindow() { return new MotifWindow(); }
  ScrollBar makeScrollBar() { return new MotifScrollBar(); }
}

In program:
  GUIFactory factory = new XXXFactory();
  w = factory.makeWindow();
  s = factory.makeScrollBar();
```

Abstract factory

- The factory class instantiates objects on behalf of the client
- from one of a family of related classes, e.g. MotifFactory
- instantiates MotifWindow and MotifScrollBar.

- New families can be introduced by providing the client with
- an instance of a new sub-class of Factory.

- The factory can ensure classes are instantiated
- consistently—e.g. MotifWindow always with
- MotifScrollBar.

- Adding a new operation involves co-ordinated change to
- the Factory class and all its sub-classes.

- ... but the problem hasn't entirely gone away: how does
- the application know which Factory to use?

- An instance of more general Strategy pattern

Adapter

Suppose you've got an existing application that accesses a data structure through the Dictionary interface:

```
public interface Dictionary {
  int size();
  boolean isEmpty();
  Object get(Object key);
}
```

... and you have a good implementation BinomialTree that instead uses another interface, say LookupTable

```
public interface LookupTable {
  int numElements();
  Object lookupKey(Object key);
}
```

The Client wishes to invoke operations on the Target interface which the Adaptee does not implement.

Adapter

Could do this:

```
public class BinomialDictionary extends BinomialTree
  implements Dictionary {

  int size()              { return numElements();        }
  boolean isEmpty()       { return (numElements() == 0); }
  Object get(Object key) { return lookupKey(key);       }
}
```

Adapter

Better:

```
public class LookupTableAdapter implements Dictionary {
  private LookupTable t;
  LookupTableAdapter(LookupTable table) {
    this.t = table;
  }

  int size()              { return t.numElements();        }
  boolean isEmpty()       { return (t.numElements() == 0); }
  Object get(Object key)  { return t.lookupKey(key);       }
}
```

The Adapter class implements the Target interface in terms of operations the Adaptee supports.

The adapter can be used with any sub-class of the adaptee

(unlike sub-classing adaptee directly).

Decorator

```
public class OutputStream {
  void write(byte[] b)
}

public class BufferedOutputStream extends OutputStream
```

Suppose we want to add a cipher capability to all streams, buffered or not:

```
public class CipheredBufferedOutputStream extends
  BufferedOutputStream

public class CipheredUnbufferedOutputStream extends
  OutputStream
```

The problem gets worse with more functionalities:

```
CheckedCipheredBufferedOutputStream
CheckedUncipheredBufferedOutputStream etc...
```

Decorator

```
interface OutputStream

class SimpleOutputStream implements OutputStream

abstract class OutputStreamDecorator implements OutputStream {
  protected OutputStream os;
  public OutputStreamDecorator(OutputStream os) {
    this.os = os;
  }
}

class BufferedOutputStream extends OutputStreamDecorator {
  write(byte[] b) {
    // do buffering
    os.write(...);
  }
}
```

To use:

```
OutputStream myStream = new CipheredOutputStream(new BufferedOutputStream(new
   SimpleOutputStream()));
```

Decorator

- The decorator pattern is an alternative to subclassing. Subclassing adds behaviour at compile time whereas decorating can provide new behaviour at runtime.
- This difference becomes most important when there are several *independent* ways of extending functionality. In some object-oriented programming languages, classes cannot be created at runtime, and it is typically not possible to predict what combinations of extensions will be needed at design time. This would mean that a new class would have to be made for every possible combination. By contrast, decorators are objects, created at runtime, and can be combined on a per-use basis. An example of the decorator pattern is the Java I/O Streams implementation.

Observer (MVC) pattern

- Suppose we have a user interface in which we need to permit the user to specify the colour of an on-screen item (like a fish in the tank...).
- The client's specification demands there be three ways to specify colours:
  - Text boxes for red, green, and blue elements;
  - Sliders for red, green, and blue elements; and
  - Selecting from a palette of commonly-used colours.
- BUT more than one of these input mechanisms might be displayed simultaneously.
- AND when the user operates one of them, the others must update appropriately.

```
class ColourTextBoxes {
  ColourSlider cs;
  ColourPalette cp;
  ...
  void changeTo(int r,int g,int b) {
    setColour(r,g,b);
    cs.update(r,g,b);
    cp.update(r,g,b);
  }
}

class ColourSlider {
  ColourTextBoxes ct;
  ColourPalette cp;
  ...
  void changeTo(int r,int g,int b) {
    setColour(r,g,b);
    ct.update(r,g,b);
    cp.update(r,g,b);
  }
}

class ColourPalette {
  ColourTextBoxes ct;
  ColourSlider cs;
  ...
  void changeTo(int r,int g,int b) {
    setColour(r,g,b);
    ct.update(r,g,b);
    cs.update(r,g,b);
  }
}
```

- Advantages
- Simple and efficient.

- Disadvantages
- If we want to add a fourth way to select colours then we have to change the code in several places.
- In general, each of the colour selection classes needs to know about all the others.

```
class Colour {
  int r,g,b;
  ColourTextBoxes ct;
  ColourSlider cs;
  ColourPalette cp;
  void setColour(int r,int g,int b) {
    this.r = r; this.g = g; this.b = b;
    ct.update();
    cs.update();
    cp.update();
  }
}

class ColourTextBoxes extends Colour {
  void changeTo(int r,int g,int b) {
    setColour(r,g,b);
  }
}

class ColourSlider extends Colour {
  void changeTo(int r,int g,int b) {
    setColour(r,g,b);
  }
}

class ColourPalette extends Colour {
  void changeTo(int r,int g,int b) {
    setColour(r,g,b);
  }
}
```

- Advantages
- Remains simple.
- Efficient.
- Easier to maintain.

- Disadvantages
- Messy—why should Colour have to know about all the sub-classes?
- Adds clutter to Colour.

```
class Colour {
  int r,g,b;
  void setColour(int r,int g,int b) {
    this.r = r;
    this.g = g;
    this.b = b;
  }
}

abstract class ColourObserver {
  ColourSubject colsubj;
  abstract void update();
}

class ColourSubject {
  Colour c;
  ColourObserver observers [];
  void setColour(int r,int g,int b) {
    c.setColour(r,g,b);
    for (int x=0;x<observers.length;++x)
      observers[x].update();
  }
  void addObserver(ColourObserver co) {
    /* insert into observers[] */
    ...
  }
}
```

```
class ColourSlider extends ColourObserver {
  void update() {
    /* read colour, redraw our GUI */
    ...
  }
  void changeTo(int r,int g,int b) {
    colsubj.setColour(r,g,b);
  }
}

class ColourTextBoxes extends ColourObserver
class ColourPalette extends ColourObserver
```

- Advantages
- Observers can be implemented as interfaces rather than as concrete classes. Doesn't use up the single opportunity to sub-class another class.
- A many-to-many, dynamically changing relationship can exist between subjects and observers.

- Disadvantages
- The flexibility limits the extent of compile-time type-checking.
- If observers can change the subject then cascading or cyclic updates could occur.
- Potential for a large amount of computational overhead. Consider slowly dragging the slider from left to right.

Visitor

```
public class Car {
  Wheel w;
  Engine e;

  Data myData;

  public int cost() {
    return someFunction(myData) + w.cost() + e.cost();
  }
}

public class Wheel {
  Data myData;
  public int cost() { return someFunction(myData); }
}

public class Engine {
  Data myData;
  public int cost() { return someFunction(myData); }
}
```

How to add a new function over the data structure?

Visitor

Alternate strategy:

```
public class CostCalculator {
  public int cost (Car c) {
    int cost = 0;

    cost += someFunction(c.getData());
    cost += someFunction(c.getWheel().getData());
    cost += someFunction(c.getEngine().getData());
  }
}
```

What's wrong with this?

Visitor

```
class CostVisitor {
  int cost = 0;
  void visit(Element e) { cost += someFunction(e); }
}

public class Car extends Element {
  Wheel wheel;
  Engine engine;

  void accept(CostVisitor v) {
    v.visit(this);

    // send visitor to children
    wheel.accept(v);
    engine.accept(v);
  }
}

public class Wheel extends Element {
  void accept(CostVisitor v) { v.visit(this); }
}
```

Visitor

Implement different visitors for different tasks, e.g.

```
abstract class Visitor {
  abstract void visit(Element e);
}

class CostVisitor extends Visitor {
  int cost = 0;
  void visit(Element e) { cost += someFunction(e.getData()); }
}

class FooVisitor extends Visitor {
  int bar = 0;
  void visit(Element e) { … someOtherFunction(e) … }
}
```

- Separate data from algorithms
- The abstract Visitor class defines operations to perform on
- each node.
- · It might perform different tasks on each different
- sub-class of Element.
- A concrete sub-class of Visitor is constructed for each kind
- of operation on the data structure.
- The methods implementing a particular operation are kept
- together in a single sub-class of Visitor.
- But changing the data structure requires changes to many
- classes.

Common themes

- Explicitly creating objects by specifying their class commits to a particular implementation.
  - It is often better to separate code responsible for instantiating objects—Abstract Factory and Singleton patterns.
- Extending functionality by subclassing commits at compile time to a particular organisation of extensions.
  - Composition and delegation may be preferable – Adapter and Decorator patterns
- Tight coupling between classes makes independent reuse difficult.
  - Separate data storage from different ways of operating on the data – Observer and Visitor patterns