OOP Spring 2005
Lecture 12
Objects and Beyond

Kasper Østerbye
IT University Copenhagen

Today's show

- What is the difference between Java and C#
- What is the difference between Java and C++

- Aspect oriented programming

- The first two OOP languages
  - Simula, 1968
  - Smalltalk, 1972,74,76,80!

Some features in C# not in Java

Most of C# is immediately recognizable to a Java programmer. Some parts are new relative to Java:

- **Virtual *and* non-virtual instance methods**
- Properties — field-style method calls
- Indexers — array-style method calls
- Operator overloading — as in C++
- Delegates — method closures
- Value types and structs — allocated on stack, inlined in objects, copied on assignment
- Reference parameters (ref and out) — much like Pascal, Ada, C++
- No inner classes, no throws clause on methods
- Unsafe code with pointer arithmetic and so on — discouraged, but possible

C#: Virtual and non-virtual instance methods

In Java, a method that is not static or private is automatically virtual.
C# has fours kinds of method declarations (the three first are known from C++):

- Static: static void M()
  A call C.M() is evaluated by finding the method M in class C or a superclass of C
- Non-virtual and non-static: void M()
  Assume that the *compile-time type* of o is T.
  A call o.M() is evaluated by finding the method M in class T or a superclass of T.
- Virtual and non-static: virtual void M()
  Assume that the *runtime class* of the object bound to o is R.
  A call o.M() is evaluated by finding the method M in class R or a superclass of R.
- Explicit interface member implementation: void I.M()
  This declaration implements M from interface I, which must be a base interface of the enclosing class.
  Assume that the *runtime class* of the object bound to o is R and its *compile-time type* is I.
  A call o.M() is evaluated by calling method I.M in class R or a superclass of R.

C# example

C#: Implicit boxing, properties, indexers, enumerators, user-defined operators, and delegates

```
public class TestLinkedList {
    static void Main(string[] args) {
        LinkedList xs = new LinkedList();
        for (int i=0; i<10; i++)
            xs.Add(i);                                  // Implicit boxing of int
        Console.WriteLine(xs.Count + " " + xs[2]);      // Property and indexer
        xs[2] = 102;                                    // Indexer's set-accessor
        int sum = 0;
        foreach (int k in xs)                           // Using an enumerator
            sum += k;                                   // and implicit unboxing
        Console.WriteLine(sum);
        LinkedList twice = xs + xs;                     // Using operator (+)
        twice.apply(new Fun(print));                    // Create delegate
    }
    static void print(object x) { Console.Write(x + " "); }
    }
    public delegate void Fun(object x); // A delegate type
```

- Most of the new devices are not essential, but they make programs shorter and clearer.
- Delegates are intended for user-interface programming (ActionListener etc), but less general than inner classes.

C++

A main difference between Java/C# and C++ is that C++ is not garbage collected.

It is the programmers responsibility to both allocate (new) and de-allocate (dispose) objects.

The ideal is:

```
...
  Person p = new Person("Lars);
  ... // use variable p only here
  p.dispose();
  // the object no longer exist!
```

But reality is:

```
  Person p = new Person("Lars);
  ... myMethod( p );
  // are there other references to
  // the object?
  p.dispose(); // is this safe?
```

The standard solution is:

```
  Person p = new Person("Lars);
  ... myMethod( p.clone() );
  // we do not give away references, but
  // copies
  p.dispose(); // is now safe
```

Cost:

creating a disposing of objects is expensive.

But if you are very careful, it can be avoided, and the programmer has better control – and can do better than GC.

Reference parameters

In Java there exist only one form of parameter kind – known as value parameters.

With a value parameter, it is the value which is passed as argument.

In C# and C++ you have to *option* to use what is known as a reference parameter.

Consider the method:

```
void foo(Integer x){
   x.value = 77;
}
```

Used as

```
Integer a = new Integer(88);
foo( a );
System.out.println( a );
```

Expected print is 88.

If foo were declared with a reference parameter:

```
void foo(ref int x){
   x = 77;
}
```

x becomes an *alias* for a.

this allows a method:

```
void swap(ref int i, ref int j){
   int temp = i;
   i = j; j = temp;
}
```

call as:

```
int a = 77, b = 88;
swap(a,b);
```

or

```
int[] ia = {1,2,3,4,5,6}
swap(ia[2], ia[5]);
```


Operator overloading

Both C# and C++ allow us to write our own versions of operators

```
class TimeOfDay{
   …
   TimeOfDay operator+(int minutes){
            …
   }
}
```

```
TimeOfDay oopStart = new TimeOfDay("14:30");
TimeOfDay break =  oopStart+45;
```

Java and reusability

The notion of
- interfaces
- abstract classes
- generics
- exceptions
- reference parameters
- operator overloading
- inner classes
- enums
- iterators
- Wrappers

are all primarily invented to cater for reusability.

If you are an application programmer,
- you will typically use interfaces, not define them,
- you will catch exceptions, not define them,
- you will use iterators, not define them etc.

Reusability – experience.

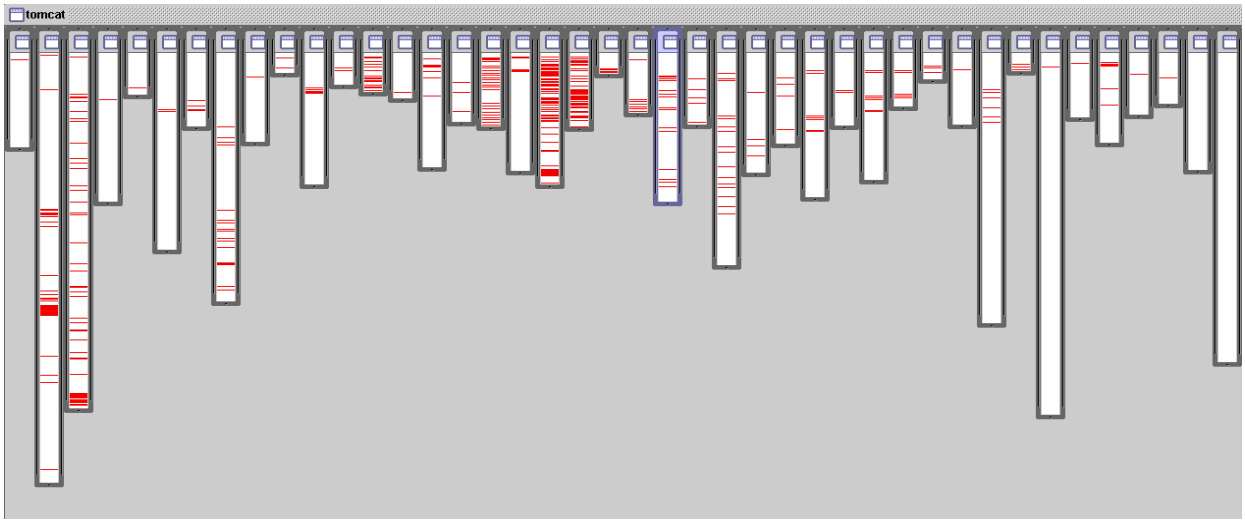
The good news:
- There is often big return when using good frameworks from Sun, MS, or other good brand.

The bad news
- It is very hard to reuse code from one project to the next. In effect, it requires you to write a framework yourself.

Aspect oriented programming

problems like…

logging is not modularized



- where is logging in org.apache.tomcat
  - red shows lines of code that handle logging
  - not in just one place
  - not even in a small number of places

problems like…

session expiration is not modularized

ApplicationSession

StandardSession

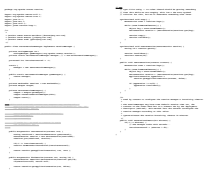
ServerSession

SessionInterceptor

StandardManager

StandardSessionManager

ServerSessionManager

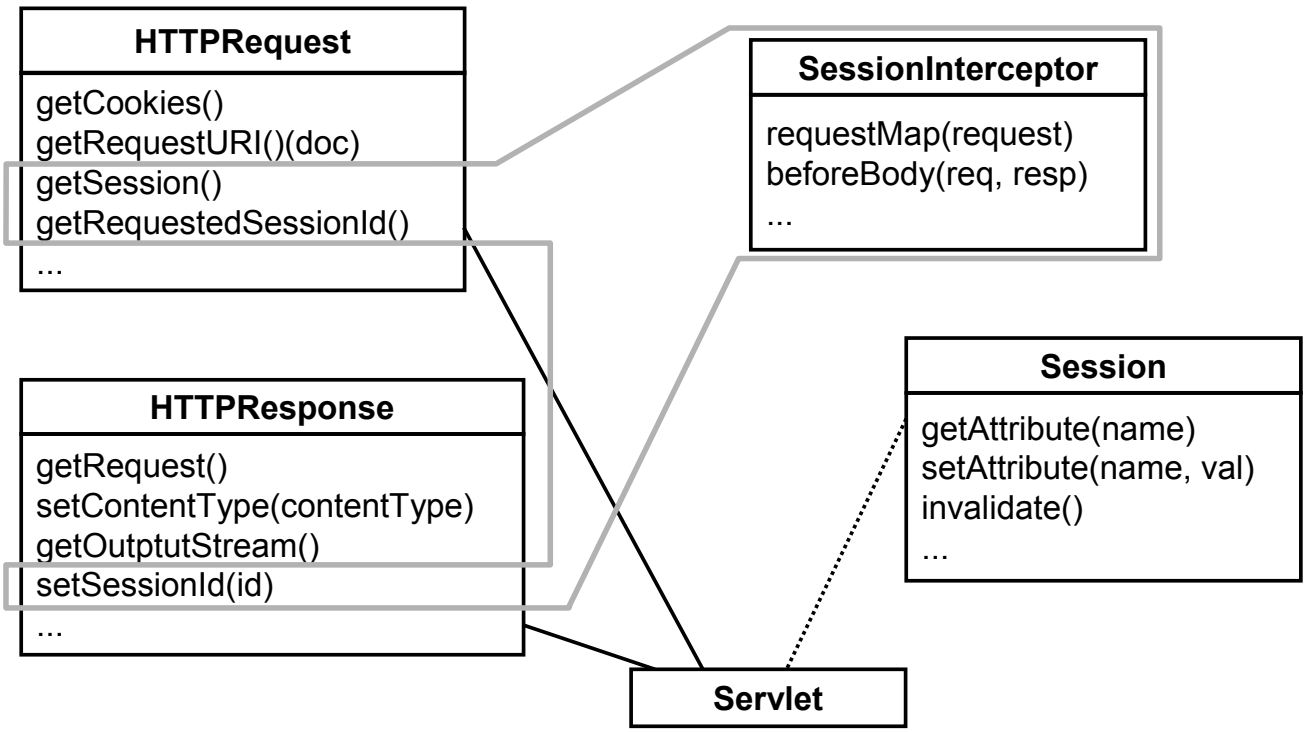


aspectj.org

the cost of tangled code

- redundant code
  - same fragment of code in many places
- difficult to reason about
  - non-explicit structure
  - the big picture of the tangling isn't clear
- difficult to change
  - have to find all the code involved
  - and be sure to change it consistently
  - and be sure not to break it by accident

crosscutting concerns



the AOP idea

- crosscutting is inherent in complex systems
- crosscutting concerns
  - have a clear purpose
  - have a natural structure
    - defined set of methods, module boundary crossings, points of resource utilization, lines of dataflow...
- so, let's capture the structure of crosscutting concerns explicitly...
  - in a modular way
  - with linguistic and tool support
- aspects are
  - well-modularized crosscutting concerns

An ultra brief overview

```
class Pip{
   private int x = 7;
}
class Pap{
   private int y = 8;
   public int getY(){ return y; }
}
```

```
privileged aspect Misc {
    public String Pip.pop(){
        return "Hello world"; }
    public int Pip.getX(){ return x; }
    public static int sum(Pip pi, Pap pa){
        return pi.getX() + pa.y;
    }

    pointcut getters():call( * *.get*() );

    before():getters(){System.out.print("*get*");}
}
```

Adds a pop method to class Pip

Adds a getX method to class Pip. Because the aspect is privileged, we can access the private fields.

A static method in the aspect. Can be used as Misc.sum(...)

A pointcut specify a number of places in the code. Here all places where the is a call to a method with a name that start with "get".

Before all getters, write "*get*".

Using the aspect

```
class AspectTest{
    public static void main(String[] a){
        Pip pip = new Pip();
        Pap pap = new Pap();
        System.out.println("Pip's pop answers " + pip.pop() );        // we can use the pop method
        System.out.println("The sum is " + Misc.sum(pip, pap) );     // The sum method is static
        System.out.println("pip.getX() is: " + pip.getX() +          // we can use the getX method
            " pap.getY() is: " + pap.getY() );    }
}
```

The output is:

```
Pip's pop answers Hello world
*get*The sum is 15
*get**get*pip.getX() is: 7 pap.getY() is: 8
```


Aspects

The goal is to put those things which belong together, but are scattered across multiple classes in one place.

Many experience that they are out of control, and can no longer find out what a class does, because new methods are added and side effects are put on the existing methods.

Simula-68

```
Class Rectangle (Width, Height); Real Width, Height;
! Class with two parameters;
 Begin
    Real Area, Perimeter;    ! Attributes;
    Procedure Update;  ! Methods (Can be Virtual);
    Begin
        Area := Width * Height;
        Perimeter := 2*(Width + Height)
    End of Update;

    Boolean Procedure IsSquare;
        IsSquare := Width=Height;
    ! Life of rectangle started at creation;
    Update;
    OutText("Rectangle created: "); OutFix(Width,2,6);
    OutFix(Height,2,6); OutImage
 End of Rectangle;
```

```
Rectangle Class LocRectangle (X, Y);
    Integer X, Y; ! More parameters;
Begin
    Boolean Tall; ! More attributes;

    Procedure Move (Dx, Dy); Integer Dx, Dy;
    Begin
        X := X + Dx; Y := Y + Dy
    End of Move;
    ! Additional life rules;
    Tall := Height > Width;
    OutText("Located at: "); OutFix(X,2,6);
    OutFix(Y,2,6); OutImage
End of LocRectangle;
```


Simula 68

- Classes
- Inheritance (called prefixing)
- Virtual methods
- Inner classes
- Inner methods
- Anonymous classes (prefixed blocks)

- Build in ”concurrency” (quasi parallism)

- No interfaces, no big library

- Simula is no longer in use.

Smalltalk 80

Same model as all the other OOP languages – objects, classes, methods, inheritance.

But!

- Everything is an object – also numbers and classes
- Everything is an object – also the code itself

- Everything is done by invoking methods on objects
  - 7 + 8 means to invoke the + method on the object 7, with the argument 8
  - (x>7) ifTrue: [Transcript show: 'That big!'] means to invoke the ifTrue method on a boolean object, with a piece of code (in [...]) as parameter.
  - Person subclass: 'Student' means to invoke the subclass method on the object Person (which represents a class). The result is an other object which represents a subclass of Person, the name of the subclass becomes 'Student'.

Smalltalk is still in use many places, it was never strong in Denmark.

Exam

- Exam date: Tuesday the 28th of June from 9-13
  (room to be announced on web)
- Question and answer session:
  - Thursday 23 june, from 16:00-18:00.
    (room to be announced on web)
- Written exam, no computers, all books and slides and notes.

- There are old exam sets and solutions. Notice:
  - Spring 2004 was too hard,
  - Fall 2004 was right.
  - Prior to 2004, it was a different teacher, different style of exam sets.

- Last chance for exercises:
  - May 10th, noon.
  - Check schema for errors.