CSE341 – Section 9
Double Dispatch, Expression Problem, Mixins, and More!

Cody Schroeder

March 7th, 2013
Outline

1. Double Dispatch
   - What? What?
   - Emulating Double Dispatch

2. Expression Problem
   - The Table
   - Examples

3. Mixins
   - Intro
   - Standard Mixins

4. Visitors
   - Visitor Pattern
Dispatch is the \textit{runtime} procedure for looking up which function to call based on the parameters given.

- What is Ruby’s procedure? (Same as Java’s)
- \textbf{Single Dispatch} on the implicit \texttt{self} parameter.
  - They use the \textit{runtime} class of the \texttt{self} parameter to lookup the correct method when a call is made.
  - This is CSE143.

\textbf{Single Dispatch} isn’t the only possible choice, though.

What about dispatching based on the \textit{runtime} classes of both \texttt{self} and the \texttt{first} method parameter?

- This is generally known as \textbf{Double Dispatch}.
  - Ruby/Java doesn’t have this, but we can emulate it.
  - This is HW7.

\textbf{Future Look}: You can dispatch on any number of the parameters and the general term for this is \textbf{Multiple Dispatch} or \textbf{Multimethods}.
The key idea to emulating double dispatch in Ruby, and on HW7, is use the built-in single dispatch procedure twice!

- Sounds simple when put that way, doesn’t it?
- Have the *principal method* immediately call another method on its *first parameter*, passing in *self*.
  - That second call will implicitly know the class of the *self* parameter.
  - It will also know the class of the *first parameter* of the *principal method* because of *Single Dispatch*.

- Of course, there are other ways to emulate double dispatch.
  - It’s often found as an idiom in SML by using case expressions.
Simple Example

```ruby
class A
  def f x
    x.fWithA self
  end
  def fWithA a
    "(a, a) case"
  end
  def fWithB b
    "(b, a) case"
  end
end

class B
  def f x
    x.fWithB self
  end
  def fWithA a
    "(a, b) case"
  end
  def fWithB b
    "(b, b) case"
  end
end

A.new.f(A.new) # "(a, a) case"
A.new.f(B.new) # "(a, b) case"
B.new.f(A.new) # "(b, a) case"
B.new.f(B.new) # "(b, b) case"
```
Simple Example (SML)

```sml
datatype t = A | B

fun f x y =
    case (x, y) of
        (A, A) => "(a, a) case"
    | (A, B) => "(a, b) case"
    | (B, A) => "(b, a) case"
    | (B, B) => "(b, b) case"
```

```
f A A; (* "(a, a) case" *)
f A B; (* "(a, b) case" *)
f B A; (* "(b, a) case" *)
f B B; (* "(b, b) case" *)
```
We have three classes \{Rock, Paper, Scissors\}

We want to write a **fight** method that returns a **winner**
between the type of **self** and another \{Rock, Paper, Scissors\}

**SML Version**

```sml
fun fight w1 w2 =
    case (w1, w2) of
    (Paper p, Rock _) => wins p
    | (Rock r, Scissors _) => wins r
    | (Scissors s, Paper _) => wins s
    | (Rock _, Paper p) => wins p
    | (Scissors _, Rock r) => wins r
    | (Paper _, Scissors s) => wins s
    | _ => tie;
```
The Expression Problem

Problem: Where do we put the code for each cell?
  - How do we group the code together?
    - By columns?? *OR* By rows??

<table>
<thead>
<tr>
<th></th>
<th>OpA</th>
<th>OpB</th>
<th>OpC</th>
<th>OpD</th>
</tr>
</thead>
<tbody>
<tr>
<td>TypeA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TypeB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TypeC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TypeD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This *can* be distilled down into an OOP vs FP argument.

- **OOP** generally groups by row (by types/classes)
  - Preferable if more likely to add types rather than operations
- **FP** generally groups by column (by operations/functions)
  - Preferable if more likely to add operations rather than types
### Rock/Paper/Scissors

<table>
<thead>
<tr>
<th></th>
<th>fight</th>
<th>to_s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rock</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Paper</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scissors</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Ruby (OOP): By rows (classes)
- SML (FP): By columns (functions)

#### lec22_stageC.rb

Same idea, just more complicated operations!
Mixins Motivation

• Look at all of these cool methods on every object!
• There seems to be a lot of recurring methods, though.
  • Is that implemented by code reuse or redundant code?
  • Maybe they have a common ancestor and use inheritance?
  • But what about String and FixNum?
    • Nearest common ancestors is Object, but Objects don’t generally have $\leq$, $<$, $\ldots$ among other methods in common.
    • Inheritance doesn’t work here, but we still want to reuse code
• Mixins are a Ruby construct that is simply for code reuse
  • Perfect for sharing code between otherwise unrelated classes

Code Examples

Sees mixins.rb.
Working with Mixins

Defining a Mixin

```ruby
module MixinNameHere
  def method1
    # do stuff
  end
  def method2(x,y,z)
    # Any arguments...
    method1  # Calling above method (ignoring shadowing)
    someOtherMethod  # This is not in the mixin
  end
end
```

Utilizing a Mixin

```ruby
class SomeClass
  include MixinNameHere
end
```
## Standard Mixins

### Comparable Mixin
- All of these methods depend on a single method named `<=>`
  - If Dan asks... say that I called it the spaceship operator.
  - It’s almost the same as `Comparable#compareTo` from Java
  - The return is restricted to the values `{−1,0,1}`

```
0 <=> 5       # -1
"ab" <=> "a"  # 1 (lexicographical ordering)
[1,2] <=> [1,2]  # 0 (analogous to Strings)
```

### Enumerable Mixin
- Awesomeness within a Module (contains 47 methods)!!!!
  - All depends on the `each` method that we’ve discussed
Visitor Pattern

- A template for handling a functional composition in OOP.
  - OOP wants to group code by classes
  - We want code grouped by functions
    - This makes it easier to add operations at a later time.
- Relies on **Double Dispatch!!!**
  - Dispatch based on \((\text{VisitorType}, \text{ValueType})\) pairs.
  - Often used to compute over AST’s (abstract syntax trees)
    - Heavily used in compilers
- Remember visitPostOrder???

**Code Examples**

See `visitor.rb` and `visitor.sml`. 