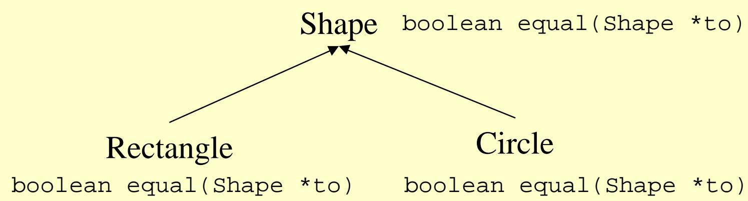
Multiple dispatching
Cecil case study

Amer Diwan

A motivating example



What does `Rectangle::equal` look like?

Rectangle::equal

- Problem: the implementation of equal really depends on the type of both “self” and the “to” argument
- Try 1:
-
```
Rectangle::equal(Rectangle *self, Shape *to) {
  if (to iskindof Rectangle) {
    Rectangle *torect := NARROW(to, Rectangle*);
    return self->top = torect->top AND
           self->left = torect->left AND ...
  }
  else return FALSE
```

- Problem: awkward if there are more type combinations to check for

Another alternative: double dispatching

-
```
Rectangle::equal(Rectangle *self; Shape *to) {
  return to->equalRect(self)
}
```
-
```
Rectangle::equalRect(Rectangle *self; Rectangle
*to) {
  return self->top = to->top AND
         self->left = to->left AND ...
}
```
-
```
Shape::equalRect(Shape *self; Rectangle *to) {
  return FALSE;
}
```

Problems with double dispatching

- Adds lots of methods, and method dispatching
- Painful to do for any more than “double” dispatching

Another alternative: multiple dispatching

- The dispatch mechanism considers all arguments when dispatching, not just the “self” argument type
- `equals(Shape *s1; Shape *s2) { return FALSE; }`
- ```
  equals(Rectangle *r1; Rectangle *r2) {
    return r1->top = r2->top AND ...
  }
  ```
- `equals(Circle *c1; Circle *c2) { ... }`
  ```

Linguistic issues with multiple dispatching

- How do you determine which multi-method to use if several can match?
- Are multi-methods inside or outside of objects
  - is it true object-orientation?
  - what are the multi-methods allowed to access?
- Some of the objections people have had to multi-methods are not dissimilar to objections to multiple inheritance

Cecil

- Cecil is a “successor” of Self
- It tries to incorporate multi-methods for the first time in a non-functional object-oriented language

Multimethod syntax in Cecil

- `equal(x@Rectangle, y@Rectangle)`
- equal is a multi-method with two constrained arguments
  - This multi-method will be invoked only if both arguments **inherit** from “Rectangle”
  - inheritance is for reuse, not for subtyping

Constrained and unconstrained arguments

- Some arguments may not be constrained
- `add_to_container(c@container, v)`
- Possibilities for arguments of a multi-method
  - no arguments constrained: an ordinary procedure
  - first argument constrained: normal singly-dispatched method
  - multiple arguments constrained: true multi-method

Conceptual view of multi-methods

- CLOS views multi-methods as being "outside" the argument types
  - What can a multi-method access?
- Cecil views multi-methods as being "inside" all objects with which an argument is constrained
  - `distance(x@Rectangle, y@Circle)` is inside both "Rectangle" and "Circle"

Implication of being "inside" the object

- Multi-methods, and particularly, automatically generated methods for accessing fields may be private
- m-m(a@A, b@B, c)
  - is "inside" A and B
  - Can therefore access private multi-methods of A and B
- Not too hard to circumvent to break encapsulation

Picking the right multi-method

- `equal(a@Shape, b@Shape) {...}`
- `equal(a@Rectangle, b@Rectangle) {...}`
- Which method is invoked in the call
  - equal(rect1, rect2)?
  - equal(rect, circle)?
  - equal(circle, circle)?

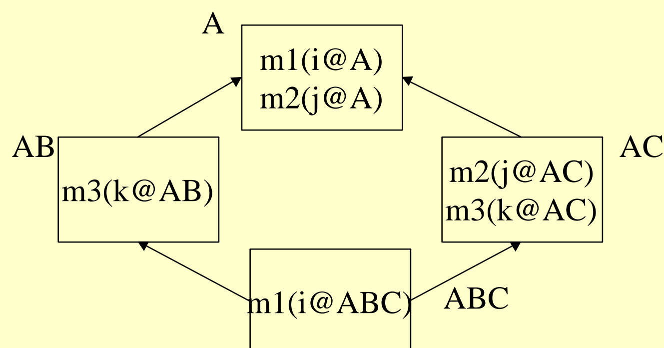
More examples

- `distance(a@Rectangle, b@Shape)`
- `distance(a@Shape, b@Circle)`
- Which multi-method is invoked in the call
  - distance(rect, circle)
- Basic idea: pick the "most specific method" for a call. If there are multiple candidates, it is an error

Cecil’s solution for conflict resolution

- Object C > Object P if C inherits from P
- Method M(m1, ..., mL) > Method N(n1, ..., nL) if for all i
  - mi is constrained and ni is not, or
  - mi and ni are both constrained and mi > ni
- On a method invocation
  - locate all multi-methods with matching name and number of arguments
  - invoke the “greatest” multi-method that matches. If no unique “greatest” method, then signal error

Examples



m1(i@ABC) > m1(i@A)
m2(j@AC) > m2(j@A)
m3(k@AB) <> m3(k@AC)

m1(abc)? m1(i@ABC)
m1(ab)? m1(i@A)
m2(abc)? m2(i@AC)
m3(abc)? error!

Analysis of resolution scheme

- Advantages
  - Simple: children override parents
  - Doesn't mask ambiguities (bugs exposed)
  - Potential conflicts can be detected before a program is run
- Disadvantages
  - Too many multi-methods may appear to conflict

Summary

- Multi-methods
  - A powerful generalization of method dispatch
- Important issues
  - Picking the "right" multi-method
  - Encapsulation

Next lecture: Final Review

- Send me email with suggestions for what you want me to review
- (reviewing everything is not an option...!)