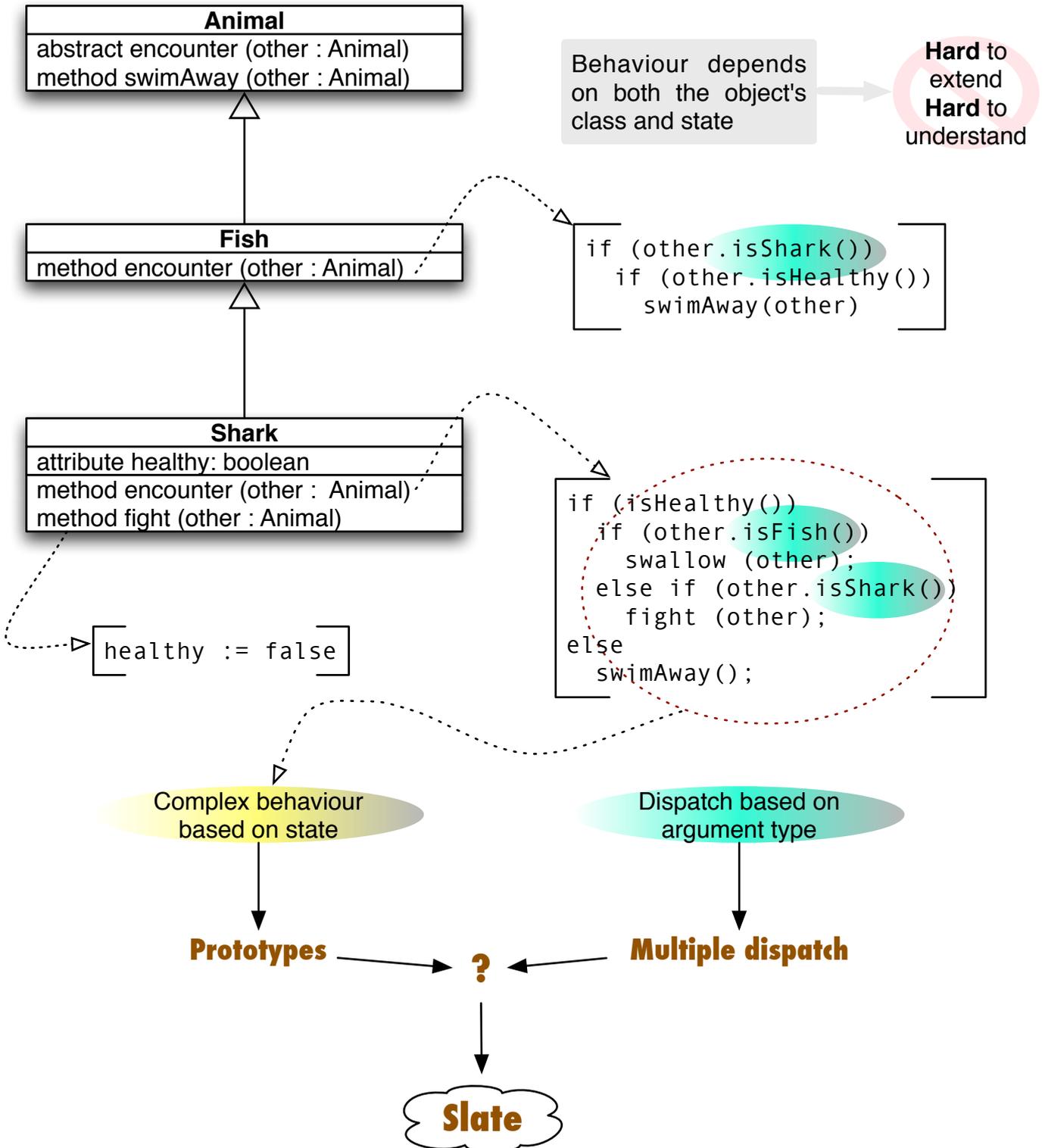


Prototypes with Multiple Dispatch: A motivation

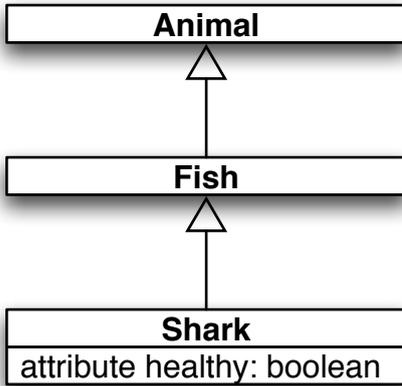
Lee Salzman and Jonathan Aldrich's Paper

Lee's presentation



Multiple dispatch

CLOS Dylan



Methods are split apart

method swimAway (a : Animal, other : Animal)

method swallow (animal : Shark, other : Animal)

method fight (animal : Shark, other : Shark)
animal.healthy := False;

method encounter (animal : Fish, other : Animal)

method encounter (animal : Fish, other : Shark)

```
if other.healthy then
    swimAway (animal, other)
```

method encounter (animal : Fish, other : Shark)

```
if other.healthy then
    swimAway (animal, other)
```

method encounter (animal : Shark, other : Fish)

```
if animal.healthy then
    swallow (animal, other)
else
    swimAway (animal, other)
```

method encounter (animal : Shark, other : Shark)

```
if animal.healthy then
    fight (animal, other)
else
    swimAway (animal, other)
```

Methods are selected taking into account **all** argument types.

Cleaner implementation logic
no conditionals on argument type

Easy extension
no modifications to existing code to introduce new subtypes

BUT

Still complex logic associated with state



Prototypes

In **single** dispatch languages:

Use the double dispatch pattern...

or libraries:

CS 338 Object-Oriented Programming
Spring Semester, 2003
Double Dispatch
Previous Lecture Notes Index Next
© 2003, All Rights Reserved, SDSU & Roger Whitney
San Diego State University - This page last updated 24 Mar 03

Contents of Doc 13, Double Dispatch
Double Dispatch
Singleton - Use Instance

References
Ralph Johnson's Object-Oriented Programming & Design lecture notes, Polymorphism (Day 5)
<http://ocw.mit.edu/ocw/csai/lectures/07/lectures.html>
VisualWorks Source Code

Doc 13, Double Dispatch Slide # 2

Double Dispatch

Multiplication Motivation

Integer * Integer
Primitive Integer * Integer operation
Integer * Float

OREILLY
ONLamp.com
LAMP THE OPEN SOURCE WEB PLATFORM

Advanced OOP: Multimethods
by David Metz
05/29/2003

Introduction

This article continues a review of advanced object-oriented programming concepts. In this installment I examine *multiple dispatch*, which is also called *multimethods*. Most object-oriented languages—including Python, Perl, Ruby, C++, and Java—are intellectually descended from Smalltalk's idea of message passing, albeit with syntax and semantics only loosely related to this source. Another option, however, is to implement polymorphism in terms of *generic functions*; this has the advantage of enabling code dispatch based on the types of multiple objects. In contrast, the native method calling style of common OOP languages only uses a single object for dispatch switching. Don't fear: this article will explain what all this means and why you should care.

With my *swt* *swt* *swt* module, Python can be extended to allow multiple dispatch. In Perl, *CLAS::DispatchMethods* serves the same purpose. Several other languages, including Java, Ruby, etc., have libraries or extensions to enable multiple dispatch. Common Lisp Object System (CLOS) and Dylan contain multiple dispatch at their heart. This article discusses an open-source implementation for the common denominator.

Dr. Carlo Pescio
Multiple Dispatch
A new approach using
templates and RTTI

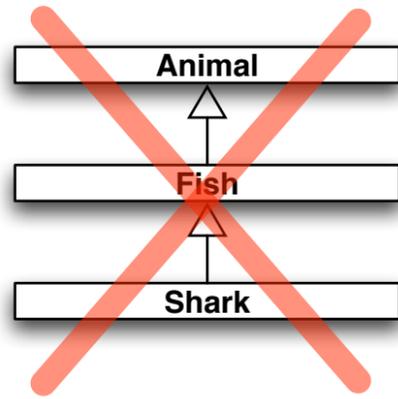
Published in C++ Report, June 1998

Virtual functions allow polymorphism on a single argument; however, sometimes there is a need for multi-argument polymorphism. Double dispatch, commonly used in C++ to implement multi-methods, does not lend easily extensible code. Solutions based on function tables are difficult to implement and prevent repeated derivation. This report focuses on two new techniques based on templates and Runtime Type Identification (RTTI). The first is faster but less flexible; the second is slower, but allows new classes to be added without any need to change existing ones.

Why multiple dispatch?
Consider an extensible hierarchy of geometric shapes. It should be easy to add new kinds of shapes or to change existing ones. It should also support several calculations on the area of the intersection of two shapes, for example:

vanilla C++ does not multi-dispatch

Prototype-based programming, the ultimate dynamic



No classes, just objects

```
object Animal
object Fish
object Shark
object HealthyShark
object DyingShark
```

Since we work with concrete instances, state can be dispensed with. There are no variables, only slots.

Objects, created on the fly, inherit behaviour slots by delegation

```
addDelegation (Fish, Animal)
addDelegation (Shark, Animal)
addDelegation (Shark, HealthyShark)
```

Methods are created and attached also on the fly

```
method Animal.swimAway () { ... }
method HealthyShark.swallow () { ... }
```

```
method HealthyShark.fight (other : Shark) {
  removeDelegation(HealthyShark);
  addDelegation(DyingShark);
}
```

```
method Fish.encounter(other) {
  if (other.isA(HealthyShark))
    swimAway();
}
method HealthyShark.encounter (other) {
  if (other.isFish())
    swallow (other)
  else if (other.isShark())
    fight (other)
}
method DyingShark.encounter (other) {
  swimAway();
}
```

Dynamic changes in delegation relations model variations in both **state** and **behaviour**.

But we re-encounter the complexity of type-based branching.

Implementations:



Self tutorial

Prototypes in Scheme

Other languages...

Prototypes + Multimethods

