Objects and Design in Smalltalk

4. Polymorphism

Éric Tanter
etanter@dcc.uchile.cl





Polymorphism

- Vital property of OOP
  - increases extensibility and clarity of code
  - relates to several ideas
    - subtyping
    - type-checking
    - dynamic binding

Dynamic Binding

"One can make a good argument on the thesis that most of the advances in both hardware and software design have been facilitated through the introduction of new late-binding mechanisms.

Going way back in hardware, we can think of index and relocation registers, memory-management units, etc.

In software, we went from absolute instruction locations and formats to symbolic assemblers, to subroutines, to relocatable code, to hardware-independent data-structure formats, to garbage collection, to the many late-binding advantages of objects, including classes and instances, message sending, encapsulation, polymorphism, and metaprogramming."

Alan C. Kay, Software: Art, Engineering, Mathematics, or Science?, Foreword in the book "Squeak: Object-Oriented Design with Multimedia Applications" by Mark Guzdial, Prentice Hall, 2001.

Dynamic Binding

- postponing selection of an operation until execution time
  - Pascal, FORTRAN, etc. do not support it
  - the compiler (or linker) compiles directions to find a given function
  - Smalltalk, C++, Java, etc. do support it
- Overloading is not dynamic binding!
  - foo(3) vs. foo("hi")
  - the compiler determines the actual function

Dynamic Binding

```
aCondition
      ifTrue: [y := 3]
      ifFalse: [y:= #(5 7 eleven)].
y printOn: someStream.
```

- many printOn: methods
  - here, the one of Integer, or the one of Array
- the Smalltalk compiler can't preordain the choice
  - the more interactive a program is, the more this is required

Misconceptions

- No, dynamic binding does not necessarily work with conditional code
- No, dynamic binding is not necessarily slow
- It all depends on the language system (compiler+runtime)
  - Smalltalk today is fast
  - C++ has techniques that make it even faster

Providing Dynamic Binding

- a technique used in non-OO languages
  - application with UI showing items in a mailbox
  - can be memos, spreadsheets, or documents

```
case y.type of
  m : showMemo(y) |
  s : showSpread(y) |
  d : showDoc(y)
end;
```

THIS IS AWFUL!!!

DON'T DO IT!!!

  - here (Pascal), type info is manually kept
  - in OO, type or class of y is available (y class)

Why not?

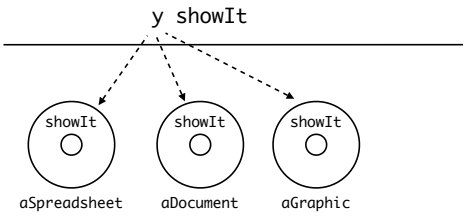
- First (less important): performance
  - conditional branches with many cases are slow
- Second (serious): engineering drawback
  - extending the application is inevitable
    - say, add Graphics to mailbox
  - need an extra case, in every client code!
    - eg. a file folder viewer
  - other operations: mail, print, save, read, etc.
    - same case statements, to be changed too
  - TEDIOUS - ERROR PRONE

What we want

- not repairing client code every time a new kind of objects is added
  - add the Graphics class, but nothing more
  - client code: `y showIt`
  - That's all!!!
- client code
  - liberated from conditionals (0)
  - immunized against a whole category of invasive alterations

Dynamic Binding enables... Polymorphism

- object-oriented design:



- each class has its own showIt method
  - an object responds directly with its own
  - the program does not know (and does not care!)

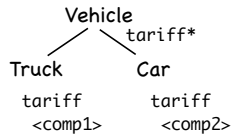
interconnection design-implementation
you can't design polymorphically unless
your language supports dynamic binding

- Poly + morphism = multiple + form
  - y assumes "multiple forms", depending on the kind of object is points to
- Dynamic binding in OO (broad sense)
  - 1. determining the object (and its type)
  - 2. looking for the method [aka. method lookup]
- Dynamic binding and polymorphism
  - dynamic binding is a mechanism (different impls)
  - polymorphism is a design technique for improving software - requires dynamic binding

Exercise

- an importer of motor vehicle
- the software must apply different calculations, depending on whether it is a truck or a car
- what should the high-level design be?
  - think of two designs

Solutions



polymorphic design

```
Vehicle
  tariff
    type = 'truck'
      ifTrue: [<comp1>]
      ifFalse: [<comp2>]
```

non-OO design

polymorphism uses classes instead of conditionals to solve problems

Smalltalk's if-less-ness

```
X:= ...
...
X ifTrue: ["true path"]
  ifFalse: ["false path"].
```

- In what class should `ifTrue:ifFalse:` be?
- What should this method do? should it contain conditionals?
- How many `ifTrue:ifFalse:` methods are there?

Summary Tip

- Think twice about conditionals
  - many are opportunities for subclassing and polymorphism
- IF statements disquiet OO programmers
- Unless you are certain never to expand them, be suspicious of conditionals
  - as you are of the GOTO in conventional prog.

On Performance

- A very peculiar topic
  - religious wars
  - what really matters is not so easy to mesure
- Effects of polymorphism in OO languages
  - 1. Fast: no conditionals
  - 2. Slow: method lookup
- Polymorphism is about cleaner systems that are more mentally tractable
  - you can't write faster code unless you can understand it
  - performance has more to do with design than language

A real example...

- CHOICES operating system
  - in C++
  - developed at University of Illinois
- First, bad performance against UNIX
- A year later, consistently superior to UNIX
  - bottlenecks were easier to fix: encapsulation
  - replacing conditionals by subclasses