

# A Use for Inheritance

Andrew P. Black

OGI School of Science & Engineering  
Oregon Health & Science University  
Portland, Oregon, USA

*blacka@ohsu.edu*



# The Rows and Columns Problem

- *“Object types and inheritance make it easy to extend the set of constructors for the type, so long as the set of operations is relatively fixed.”*
  - constructors = classes = representations
- *“Algebraic types and [pattern] matching make it easy to add new operations over the type, so long as the set of constructors is relatively fixed.”*

[Odersky & Wadler, POPL'97]

- operations = messages = observers



# The List Example

- 2 representations, each understanding 3 messages
  - 6 methods
  - Matrix of Operations and Constructors

		Operations		
		first	rest	isEmpty
Representation	ConsList (e, l)	$\wedge e$	$\wedge l$	false
	EmptyList ()	error	error	true



# The List Example

- 2 representations, each understanding 3 messages
  - 6 methods
  - Classes are represented by rows from the matrix

		Operations			
		first	rest	isEmpty	
Representation	ConsList (e, l)	$\wedge e$	$\wedge l$	false	ConsList class
	EmptyList	error	error	true	EmptyList class



# The List Example

- 2 representations, each understanding 3 messages
  - 6 methods
  - Functions are represented by columns from the matrix

		Operations		
		first	rest	isEmpty
Representation	ConsList (e, l)	$\wedge e$	$\wedge l$	false
	EmptyList	error	error	true

*first function*

*rest function*

*isEmpty function*

# Summary of the Problem

- O-O (co-inductive) definitions make it
  - easy to add new rows (classes)
  - hard to add new columns (operations)
    - each new operation must be defined on *every* row
    - this means editing *every* class that implements the abstraction
- ADT (inductive) definitions make it
  - easy to add new columns (observers)
  - hard to add new rows (constructors)
    - observers defined exhaustively by cases over constructors
    - adding a constructor required editing every observer



# A Solution

## Use Inheritance!

- A pattern called the core/support split
- Should be applied when several classes implement the same abstraction in different ways
- Key idea: separate the methods into:
  - a *core* set that is capable of observing *all* off the distinctions between objects
  - a *support* set that implements the useful utility functions



# The Core Methods

- Core Methods access the implementation details of the various classes
  - direct access to the instance variables
- Core methods are private to the particular implementation of the abstraction
- For lists, *first*, *rest* and *isEmpty* form the core set

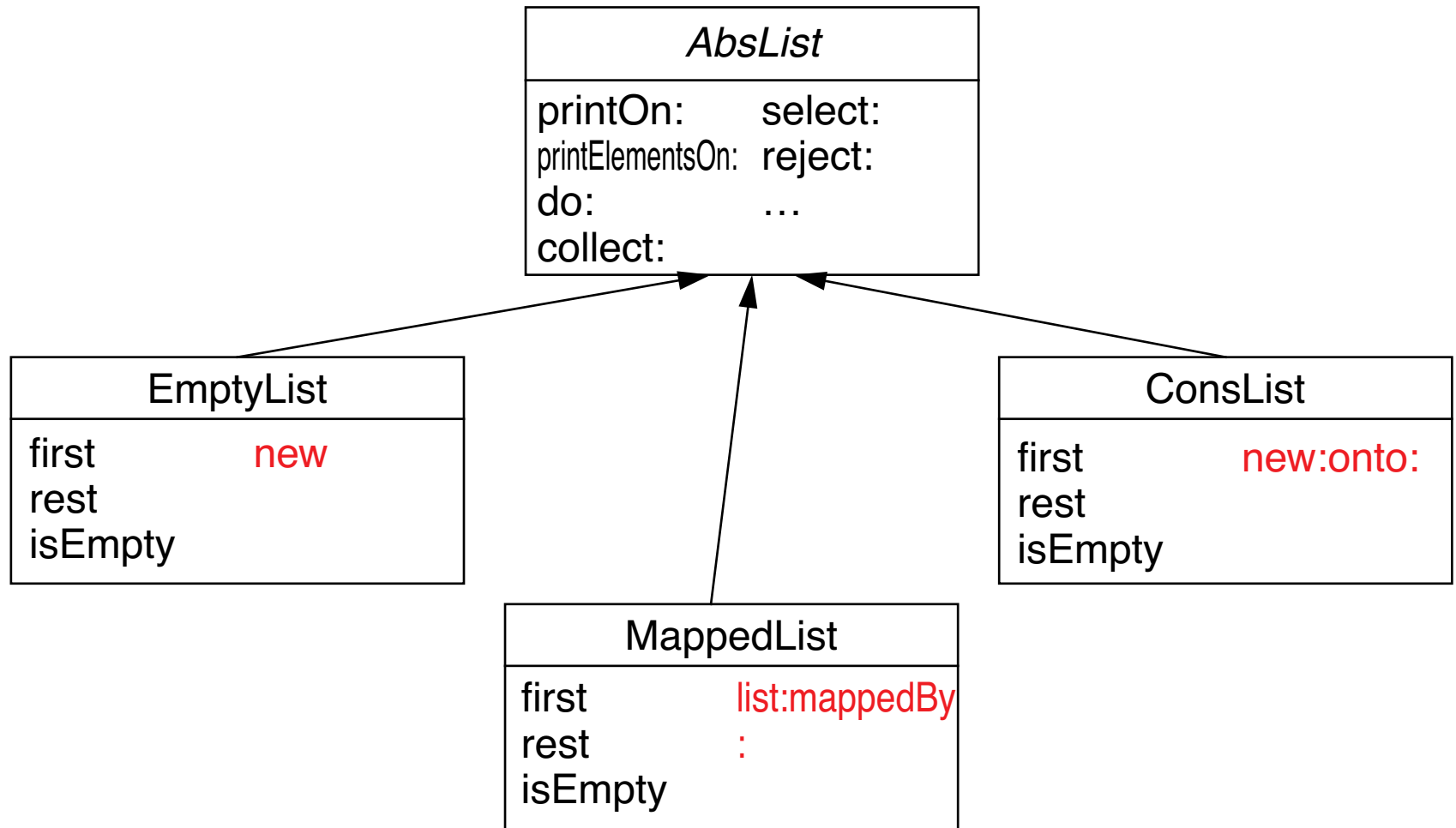


# The Support Methods

- Support Methods do not access any of the implementation details of the various classes
  - *no access* to the instance variables
  - *all* information about the object obtained through use of a core method
    - if no core method tells you what you need, then the core set must be expended
- Support methods are promoted to an abstract class that is a superclass of all of the implementation classes
- For lists, *printString*, *printOn: do: select:* and *collect:* are all in the support set



# The Result



# Other Examples

- Magnitude in Smalltalk

=, < and *hash* form the core set

*max:*, *min:*, *between:and:* are support methods

- Stream in Smalltalk

*next*, *atEnd* and *nextPut:*, also *contents* for efficiency

16 to 30 support methods depending on dialect

- `java.io.Reader`

core methods are *read(char[], int, int)* and *close()*

some subclasses (*BufferedReader*, *CharArrayReader*,  
*FilterReader*, *InputStreamReader*, *PipedReader*, *StringReader*)

re-implement other methods for efficiency.



# Related Work

- Wirfs-Brock [Prentice Hall 1990]
- Template pattern [GoF, 1995]
- Abstract Class [Woolf, 1997]
- Lamping [OOPSLA'93]

