Applying Traits to the Smalltalk Collection Classes

Andrew P. Black

Portland State University, Portland, Oregon

Joint work with
Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz
Software Composition Group, University of Bern, Switzerland
A Play in Two Acts

- Act I
  - Scene 1, in which we are introduced to the Smalltalk collection classes, and the problems of code reuse therein
  - Scene 2: enter the *trait*, a solution to those problems

- Act II
  - Scene 1, in which *traits* are used to vanquish the wicked and set all right in the world
Act I

Scene 1: The Smalltalk Collection Classes
The Smalltalk Collection Classes

- General-purpose library classes
  - “Blue Book” (1983) classes include
    - 19 subclasses of Collection
    - 9 subclasses of Stream
  - Squeak (2002) classes include
    - 98 subclasses of Collection, of which 39 are “Collections”
    - 39 subclasses of Stream, of which 10 are “Collections”
      - Bitmap, FileStream, CompiledMethod etc. are application-specific subclasses, not part of the collections library
Protocols in the Collection Classes

- All collection objects support a uniform set of well-defined protocols.
  - Testing protocol: `isEmpty`, `includes:`, `occurrencesOf`:
  - Enumeration protocol: `do:`, `select:`, `collect:`, `reject`:
  - Copying protocol: `copy`, `copyWith:`, `copyWithout`:

- Without uniform protocols:
  - client code depends on the class of the collection
The Varieties of Collection

Beyond this uniformity, there are many different kinds of collection

- **Array** (indexable, fixed size)
- **OrderedCollection** (indexable, extensible)
- **Interval** (indexable, immutable)
- **SortedCollection, Heap** (indexable, extensible, sorted)
- **Set** (unordered, no duplicates)
- **Bag** (unordered, duplicates allowed)
- *... and more*
Classifying the Collections

The differences between the collections manifest themselves in several dimensions

- Order: unordered (Set), explicitly ordered (LinkedList), or implicitly ordered (SortedCollection)
- Extensibility: fixed size (Array) or variable size (Heap)
- Mutability: immutable (Symbol, Interval) or mutable
- Duplicates: allowed (Bag) or disallowed (Set)
- Comparison: using identity (IdentitySet) or using the object-level equality operator (Set)

etc.
Implementation
The implementation of the Smalltalk collection hierarchy ...
- has been improved over more than 20 years
- is often considered a paradigmatic example of object-oriented programming
Implementation

- The implementation of the Smalltalk collection hierarchy ...
  - has been improved over more than 20 years
  - is often considered a paradigmatic example of object-oriented programming
- But: the implementation is still far from optimal!
The implementation of the Smalltalk collection hierarchy ...

- has been improved over more than 20 years
- is often considered a paradigmatic example of object-oriented programming

But: the implementation is still far from optimal!

Single inheritance is not powerful enough to model a hierarchy of classes that can be categorized in so many dimensions
Classification: the Problem
Classification: the Problem

Collection
Classification: the Problem

Collection

Order
Classification: the Problem

Order

Collection

Sequenced

Unsequenced

Order
Classification: the Problem

Collection

Sequenced

Unsequenced

Order

Extensibility
Classification: the Problem

Collection

Order

Sequenced

Unsequenced

Extensibility

Fixed
Classification: the Problem

Collection

Sequenced

Unsequenced

Fixed

Variable

Order

Extensibility
Classification: the Problem

Collection

Sequenced

Unsequenced

Order

Fixed

Variable

Extensibility

Array
Classification: the Problem

- **Collection**
  - **Sequenced**
    - **Fixed**
      - **Array**
    - **Variable**
      - **Ordered-Collection**
  - **Unsequenced**

**Order**: Fixed, Variable

**Extensibility**: Collection
Classification: the Problem

Collection

Sequenced
- Fixed
  - Array
- Variable
  - Ordered-Collection

Unsequenced

Order

Extensibility
- FixedSet

Applying Traits to the Smalltalk Collection Classes
Classification: the Problem

- Collection
  - Sequenced
    - Fixed
      - Array
    - Variable
  - Unsequenced
    - Fixed
      - FixedSet
    - Ordered-Collection

Order
Extensibility
Classification: the Problem

- Collection
  - Sequenced
    - Fixed
      - Array
    - Variable
      - Ordered-Collection
  - Unsequenced
    - Fixed
      - FixedSet
    - Unsequenced
      - Set

Order

Extensibility
Classification: the Problem

Order

Collection

Sequenced

Fixed

Variable

Array

Ordered-Collection

Unsequenced

Fixed

Variable

FixedSet

Set

Extensibility

Applying Traits to the Smalltalk Collection Classes
Classification: the Problem

Single inheritance is not up to the job
Consequences

- As a consequence, the Squeak collection hierarchy suffers from:
  - Methods implemented “too high”
  - Code duplication
  - Unnecessary inheritance
  - Deficiencies in Modeling
Methods implemented “too high”

Collection

Sequenceable-Collection

Set
  add:
  addAll:
  remove:
  removeAll:

Heap
  add:
  addAll:
  remove:
  removeAll:

Array
Methods implemented “too high”

Collection

```
Set
  add:
  addAll:
  remove:
  removeAll:

Heap
  add:
  addAll:
  remove:
  removeAll:

Array
```

Sequenceable-Collection
Methods implemented “too high”

![Diagram of Collection, Sequenceable-Collection, Set, Heap, Array]
Methods implemented “too high”

```
Collection
  add:
  addAll:
  remove:
  removeAll:

Sequenceable-Collection

Set
Heap
Array
```
Methods implemented “too high”

```
Collection
  add:
  addAll:
  remove:
  removeAll:

Sequenceable-Collection
  Set
  Heap
  Array
    add:
    addAll:
    remove:
    removeAll:
```

Applying Traits to the Smalltalk Collection Classes
Methods implemented “too high”

- **Collection**
  - add:
  - addAll:
  - remove:
  - removeAll:

- **Sequenceable-Collection**
  - add: anObject
  - self shouldNotImplement

- **Set**
- **Heap**
- **Array**
  - add:
  - addAll:
  - remove:
  - removeAll:
Methods implemented “too high”

```
Collection
  add:
  addAll:
  remove:
  removeAll:

addAll: aCollection
  ↑ aCollection do: [:each | self add: each ]

Sequenceable-Collection

Set

Heap

Array
  add:
  addAll:
  remove:
  removeAll:

add: anObject
  self shouldNotImplement
```
Methods implemented “too high”

```
Collection
add:
addAll:
remove:
removeAll:

addAll: aCollection
   ^ aCollection do: [ : each | self add: each ]

Sequenceable-Collection

Set
Heap

Array
add:
remove:
removeAll:

add: anObject
   self shouldNotImplement
```
Methods implemented “too high”
In the Collection Classes, we found
- 131 methods (more than 10%) are implemented too high
- 106 methods (80%) are *not* explicitly cancelled

Consequences
- Hard to see the *real* interface of a class
- Unexpected runtime errors
  - when an inappropriate method is called
Code Duplication

- Alternative to implementing methods “too high”
- Often used when implementing methods “too high” is hard or impossible
Unnecessary Inheritance

- Inheritance is often used when only a few of the superclass methods should be inherited.
- Many methods are inherited unnecessarily:
  - Makes understanding a class much harder.
  - Instead of improving understandability, this form of inheritance hinders understanding of the code.
- Example: Dictionary inherits from Set, but presents a very different interface:
  - Methods such as remove:, remove:ifAbsent: need to be disabled.
Deficiencies in Modeling

- For **Sets** and **Dictionaries** there are different classes that exhibit different ways of comparing elements
  - **Set** (uses method =)
  - **IdentitySet** (uses method ==)
  - **PluggableSet** (uses “pluggable” comparison block)

But:
- Corresponding variants are not available for other classes such as **Bag**, **Heap**, etc.
- Implementing them is cumbersome because there is no way to factor-out and reuse “pluggability”
Missing Properties

- Other properties such as immutability are not captured in a general way
  - Two ad-hoc implementations in `Interval` and `Symbol`
- The interfaces of some classes are missing methods that one would expect:
  - `String` adds 142 methods to the protocol of its superclass (searching for substrings, regular expressions)
  - `Text`, which represents a string with visual attributes, implements only 15 of them. The other 127 are missing!
But that’s not all!

- Suppose that one would like to implement collection-like behavior *outside* of the collection hierarchy
  - *Path*, a subclass of *DisplayObject*, contains an ordered sequence of points
  - It would be useful to be able to do: and collect: over them
- How can we re-use code from the Collection Classes?
  - We can copy and paste it!
Wouldn’t it be nice if...

Hierarchy of graphical objects

GraphicalObject

Circle

Polygon

Generic properties

Colored

Bordered

SyncDrawing
Wouldn’t it be nice if...

Hierarchy of graphical objects

Generic properties

GraphicalObject

Circle

Polygon

SyncColoredCircle

Colored

Bordered

SyncDrawing
Wouldn’t it be nice if...

Hierarchy of graphical objects

- GraphicalObject
  - Circle
  - Polygon
  - SyncColoredCircle
  - BorderedColoredPolygon

Generic properties

- Colored
- Bordered
- SyncDrawing

29th October 2004  Applying Traits to the Smalltalk Collection Classes
Wouldn’t it be nice if...

Hierarchy of graphical objects

- GraphicalObject
  - Circle
  - Polygon
    - SyncColoredCircle
      - SyncDrawing
    - BorderedColoredPolygon

Generic properties

- Colored
- Bordered
- SyncDrawing
Wouldn’t it be nice if...

Hierarchy of graphical objects

Generic properties

GraphicalObject

Circle

Polygon

SyncColoredCircle

SyncDrawing

Colored

BorderedColoredPolygon

Colored

Bounded

SyncDrawing
Wouldn’t it be nice if...

Hierarchy of graphical objects

```
GraphicalObject

Circle

SyncColoredCircle
  SyncDrawing
  Colored

Polygon

BorderedColoredPolygon
  Bordered
  Colored
```

Generic properties

- Colored
- Bordered
- SyncDrawing
Wouldn’t it be nice if...

Hierarchy of graphical objects

- GraphicalObject
  - Circle
  - Polygon
    - SyncColoredCircle
      - SyncDrawing
      - Colored
    - BorderedColoredPolygon
      - Bordered
      - Colored

Generic properties

- Colored
- Bordered
- SyncDrawing
Existing reuse technologies are inadequate

- Single inheritance is not expressive enough
- Language designers proposed alternatives
  - Multiple inheritance
  - Mixin inheritance
- However, both of these approaches have drawbacks
  - …but that’s another talk [Schärli et al, ECOOP 2003]
Scene 2: Enter *Traits*

which solve all of these problems, without the drawbacks
Goals for Traits

- Improve code reuse
- Maintain understandability
- No surprises
- Avoid fragile hierarchies
What are Traits?

- Traits are parameterized behaviors
  - Traits provide a set of methods (● → )
  - Traits require a set of methods ( → )
  - Traits are purely behavioral
    - just methods, no instance variables (no state)

<table>
<thead>
<tr>
<th>TCircle</th>
</tr>
</thead>
<tbody>
<tr>
<td>● area</td>
</tr>
<tr>
<td>● bounds</td>
</tr>
<tr>
<td>● diameter</td>
</tr>
<tr>
<td>● hash</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
How are Traits Used?
How are Traits Used?

- Traits are the behavioral building blocks for classes
How are Traits Used?

- Traits are the behavioral building blocks for classes
  - \( Class = \)
How are Traits Used?

- Traits are the behavioral building blocks for classes
  - $\text{Class} =$

```
ColoredCircle
```

29th October 2004 Applying Traits to the Smalltalk Collection Classes
How are Traits Used?

- Traits are the behavioral building blocks for classes
  
  - *Class* = *Superclass*

```
Class = Superclass
```

![Diagram](image_url)
How are Traits Used?

- Traits are the behavioral building blocks for classes
  - \( \textit{Class} = \textit{Superclass} + \textit{State} \)
How are Traits Used?

- Traits are the behavioral building blocks for classes
  - \( \text{Class} = \text{Superclass} + \text{State} + \text{Traits} \)

![Diagram showing the relationship between Object, ColoredCircle, TColor, and TCircle.](image)
How are Traits Used?

- Traits are the behavioral building blocks for classes
  
  \[\text{Class} = \text{Superclass} + \text{State} + \text{Traits} + \text{Glue methods}\]
How are Traits Used?

- Traits are the behavioral building blocks for classes
  - $\text{Class} = \text{Superclass} + \text{State} + \text{Traits} + \text{Glue methods}$
How are Traits Used?

- Traits are the behavioral building blocks for classes
  - \( \text{Class} = \text{Superclass} + \text{State} + \text{Traits} + \text{Glue methods} \)

- Traits do not replace single inheritance
  - Traits provide modularity within classes
Composition Rules

Class methods take precedence over trait methods
Composition Rules

- Class methods take precedence over trait methods
Composition Rules

- Class methods take precedence over trait methods
Composition Rules

- Class methods take precedence over trait methods

```
ColoredCircle

TCircle
  draw
  draw
  radius

TColor
  hue
  rgb
```

29th October 2004

Applying Traits to the Smalltalk Collection Classes
Conflicts Must be Resolved *Explicitly*

---

**Diagram:**

```
<table>
<thead>
<tr>
<th>ColoredCircle</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCircle</td>
</tr>
<tr>
<td>hash</td>
</tr>
<tr>
<td>TColor</td>
</tr>
<tr>
<td>hash</td>
</tr>
</tbody>
</table>
```
Conflicts Must be Resolved *Explicitly*
Conflicts Must be Resolved *Explicitly*
Conflicts Must be Resolved *Explicitly*
Conflicts Must be Resolved *Explicitly*
Conflicts Must be Resolved *Explicitly*
Conflicts Must be Resolved *Explicitly*
Conflicts Must be Resolved *Explicitly*

```
hash
↑ self colorHash bitXor:
  [self circleHash]
```

![Diagram showing the relationships between classes and their properties.](image-url)
Conflicts Must be Resolved *Explicitly*

I) Override the conflict with a glue method
Conflicts Must be Resolved *Explicitly*

I) Override the conflict with a glue method
   - Aliases provide access to the conflicting methods
Conflicts Must be Resolved *Explicitly*

I) Override the conflict with a glue method
   - Aliases provide access to the conflicting methods
I) Override the conflict with a glue method
   - Aliases provide access to the conflicting methods
Conflicts Must be Resolved *Explicitly*

I) Override the conflict with a glue method
   - Aliases provide access to the conflicting methods
Conflicts Must be Resolved *Explicitly*

I) Override the conflict with a glue method
   - Aliases provide access to the conflicting methods
Conflicts Must be Resolved *Explicitly*

<table>
<thead>
<tr>
<th>ColoredCircle</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tcircle</strong></td>
</tr>
<tr>
<td><strong>h</strong></td>
</tr>
<tr>
<td><strong>Tcolor</strong></td>
</tr>
<tr>
<td><strong>h</strong></td>
</tr>
</tbody>
</table>

1) Override the conflict with a glue method
   - Aliases provide access to the conflicting methods
Conflicts Must be Resolved *Explicitly*
Conflicts Must be Resolved *Explicitly*

II) Avoid the conflict
- Edit one of the traits
- Exclude one of the conflicting methods from the composition
Conflicts Must be Resolved *Explicitly*

II) Avoid the conflict
- Edit one of the traits
- Exclude one of the conflicting methods from the composition
Conflicts Must be Resolved \textit{Explicitly}

\begin{itemize}
  \item \textit{Avoid the conflict}
    \begin{itemize}
      \item Edit one of the traits
      \item Exclude one of the conflicting methods from the composition
    \end{itemize}
\end{itemize}
Conflicts Must be Resolved *Explicitly*

**II) Avoid the conflict**
- Edit one of the traits
- Exclude one of the conflicting methods from the composition
Conflicts Must be Resolved *Explicitly*

II) Avoid the conflict
- Edit one of the traits
- Exclude one of the conflicting methods from the composition
Conflicts Must be Resolved *Explicitly*

II) Avoid the conflict
- Edit one of the traits
- Exclude one of the conflicting methods from the composition
Flattening Property (Two views on code)

Structured view

![Diagram showing the structure of ColoredCircle class with properties and methods related to drawing a colored circle.]

- **Object**
- **ColoredCircle**
- **draw**
- **TCircle**
  - draw
  - radius
- **TColor**
  - hue
  - rgb

29th October 2004

Applying Traits to the Smalltalk Collection Classes
Flattening Property (Two views on code)

Structured view

Object

ColoredCircle

draw

TCircle
draw
radius

TColor
hue
rgb

Object

ColoredCircle

draw
radius
hue
rgb
Flattening Property (Two views on code)

Structured view

Object

ColoredCircle

draw

TCircle

draw
radius

TColor

hue
rgb

Flat view

Object

ColoredCircle

draw
radius
hue
rgb
Flattening Property (Two views on code)

**Structured view**

- **Object**
  - **ColoredCircle**
    - **draw**
    - **TCircle**
      - **draw**
      - **radius**
    - **TColor**
      - **hue**
      - **rgb**

**Flat view**

- **Object**
  - **ColoredCircle**
    - **draw**
    - **radius**
    - **hue**
    - **rgb**

**equivalent**
Composition of multiple properties

![Diagram showing the composition of multiple properties]

Circle

BorderedColoredCircle

TBordered

TColored
Composition of multiple properties

- “Live” illustration
  - Traits implementation on top of *Squeak* Smalltalk
Conceptual Evaluation of Traits

- Improve code reuse
  - Composing classes from reusable building blocks
- Maintain/Improve understandability
  - Flattening property → Two views on the code
    - Flat View – Ordinary Smalltalk (as if no traits were used)
    - Structured View – Separating glue methods from reusable behavior
- No surprises
  - Simple composition rules
  - Explicit conflict resolution
- Avoid fragile hierarchies
  - Composite entity is in full control
Current Status

- Fully implemented in the Smalltalk dialect *Squeak*
  - No performance penalty for method lookup
  - Indirect instance variable access
- Used to refactor the Smalltalk collection hierarchy
- Used to build reusable metaclass properties
- Used to rebuild the trait implementation!
- Formal model
Act II

Scene 1: Using Traits to Refactor the Collection Classes
Refactoring Strategy

- Create traits for the identified collection properties
  - Functional properties
    (e.g., extensible, explicitly ordered, implicitly ordered)
  - Implementation properties
    (e.g., array based implementation, linked implementation)
- Combine them to build the required collection classes
Refactoring Process

- Iterative and “bottom up”
- Basic steps:
  - Pick an existing class (e.g., SequenceableCollection)
    - Create a new (empty) trait for one aspect of its behaviour (e.g., TEnumeration)
    - Make the class use this new trait and move the corresponding methods into the trait
  - Iterate...
Refactoring Process (2)

- Refinement steps:
  - Making the traits more fine-grained
    - Breaking traits into multiple subtraits
  - Juggling with classes and traits
    - Renaming classes and traits
    - Introducing new abstract classes
    - Replacing inheritance with trait composition
Resulting Class Hierarchy

- Resulting class hierarchy consists of 3 parts
  - Abstract root class **Collection**
    - contains only those methods supported by *all* collection classes
  - Layer of abstract classes providing the public *functionality* of the different kinds of collection
  - Layer of concrete collection classes that:
    - inherit the public functionality from one of the functionality classes
    - use a trait that adds a specific *implementation*
Resulting Trait Hierarchies

- Two trait hierarchies
  - Functional Traits
  - Implementation Traits
- Very fine-grained
  - Most traits consist of multiple subtraits
Functional Traits
Let’s summarize...

- We refactored 30 concrete and 8 abstract classes
  - 29 subclasses of Collection
  - 9 subclasses of Stream
- The refactored classes are built from 67 traits
  - The average number of traits used to build a class is more than 5
  - The maximum number of traits per class is 22
What did we gain?

- Improved Understandability
  - No unnecessary inheritance
    - Every inheritance relationship is conceptually sound
    - No abuse of inheritance for the purpose of reusing only a fraction of the superclass’ methods
  - No methods implemented “too high”
    - Every class has a clean interface

- More Uniform Protocols
  - No missing methods
    - Classes provide all the methods in the appropriate protocols
    - Collections can be used more uniformly
What did we gain? (2)

- Less code!
  - Elimination of code duplication
    - Eliminates potential “replicated bug” problems
  - No need to disable methods that are implemented “too high”
  - In numbers:
    - About 10% less source code
    - More than 20% fewer methods
What did we gain? (3)

- Improved reusability
  - The “toolbox of traits” makes it much easier to create new collection classes
    - *e.g.*, a class `PluggableBag` can be created as a subclass of `Bag` by using the trait `TPluggableAdaptor`
  - Traits can be reused outside of the collection hierarchy
    - *e.g.*, the trait `TEnumeration` can be used to add the enumeration protocol to the class `Path`

```plaintext
newPath := path collect:
    [:each | rect containsPoint: each]
```
Why Traits (vs. Mixins and Multiple Inheritance)?

- Main reason: flattening property
  - Although classes are built from up to 22 traits, we can still view and work with these classes in the ordinary (i.e., single inheritance) way
  - There is no trade-off between fine grained components and understandability
    - We introduced more traits than were strictly necessary in order to improve understandability and unanticipated reuse
- This is not so for mixins and multiple inheritance
Why Traits (vs. Mixins and Multiple Inheritance)? (2)

Traits

Collection

Extensible-Sequenced-Explicitly

Ordered-Collection
Why Traits (vs. Mixins and Multiple Inheritance)? (2)

Traits

Collection

Extensible-Sequenced-Explicitly

Ordered-Collection

Mixins

Collection

22

Ordered-Collection
Why Traits (vs. Mixins and Multiple Inheritance)? (2)

- **Traits**
  - Collection
  - Extensible-Sequenced-Explicitly
  - Ordered-Collection

- **Mixins**
  - Collection
  - Ordered-Collection

- **Multiple Inheritance**
  - Collection
  - Ordered-Collection
  - 22
Related Work on Refactoring, Reuse and Collections

- William Cook’s study of conformance and inheritance in the Smalltalk-80 collection classes
- Other languages with more sophisticated collection hierarchies
  - Self (also using a construct called “traits”)
  - Eiffel (using multiple inheritance)
  - Animorphic Smalltalk (using mixins)
  - these approaches are all coarse-grained compared to traits
- A lot of work on refactoring
  - Semi-automated refactoring
Related Work on Languages

- Xerox Star project in Mesa [Curry ’82]
  - really an MI framework
- Larch specification language [Guttag ’85]
  - adding a trait constrains behavior of methods
- Jigsaw modularity framework [Bracha ’92]
  - completeness vs. simplicity
- Caesar [Mezini ’97, ’02]
  - Combination layer between objects and classes
- Logtalk [Unpublished]
  - reusable categories, but no composition of categories
Related Work at OGI [WoDiSEE ’04]

- Multiview: multiple views on a program
  - moving away from linear text to more structured and expressive views of programs
- Refactorings considered as defining equivalence relations …
  - programs as the equivalence classes that they induce
- As the language becomes more expressive:
  - more refactorings become available
  - more equivalence relations exist
  - more views are can be utilized
Limitations

- Unintended name captures
  - This is a limitation of Smalltalk
    - No visibility/scoping mechanism for methods
  - We are working on an extension of Smalltalk/Traits that tackles this problem

- Traits cannot specify state
  - Slots and accessor methods are automatically generated
Conclusion

- Traits have nice theoretical properties
- These properties of traits paid off in practice!
  - Flattening property
  - Sum operation with explicit conflicts
- Migrating to traits is easy
  - No change in method level syntax/semantics
  - Ordinary Smalltalk programmer can understand, use, and work with the new hierarchy
- Fine-grained trait structure has no disadvantages
- Refactoring with traits was fun!
  - Tools are important
Thank you

Questions?