Applying Traits to the Smalltalk Collection Classes

Andrew P. Black

OGI School of Science and Engineering, Portland, USA

Nathanael Schärli, Stéphane Ducasse

Software Composition Group, University of Berne, Switzerland

Roadmap

- The Smalltalk Collection Classes
  - Overview
  - Problems of the Single Inheritance Implementation
- Traits
- Using Traits to Refactor the Collection Classes
  - Strategy and Process
  - Results and Discussion
- Related Work and Conclusion

The Smalltalk Collection Classes

- Loosely defined group of general purpose subclasses of `Collection` and `Stream`
- The "Blue Book" specifies the core of the hierarchy
  - 17 subclasses of `Collection`
  - 9 subclasses of `Stream`
- Recent Smalltalk systems such as *Squeak* contain more classes. For our study, we considered...
  - 37 subclasses of `Collection`
  - 10 subclasses of `Stream`

Programming with Collections

- Each collection object supports a set of well-defined protocols
  - Testing methods: `isEmpty`, `includes:`, `occurencesOf:`
  - Enumeration methods: `do:`, `select:`, `collect:`, `reject:`
  - Copying methods: `copy`, `copyWith:`, `copyWithout:`
  - *etc.*
- Programming with aggregates rather than individual elements
  - Significantly raises level of abstraction!

Programming Examples

- With *any* collection object we can do:
  - Filtering into new collection

```
result := students select: [:each | each gpa < 3.5].
```

  - Accumulate values for each element and obtain average

```
sumGPA := students inject: 0 into:
                [:sum :each | sum + each gpa].
avgGPA := sumGPA / students size.
```

- Without uniform collection classes
  - Program code depends on the type of collection
  - Loops are necessary

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)
  - *etc.*

Classifying the Collections

- The differences between the collections manifest themselves in several different dimensions
  - Order: Unordered (Set), explicitly ordered (LinkedList), or implicitly ordered (SortedCollection)
  - Extensibility: Fixed size (Array) or variable size (Heap)
  - Mutability: Immutable (String) or mutable (the others)
  - Duplicates: Allowed (Bag) or disallowed (Set)
  - Comparison: Using identity (IdentitySet) or using a higher-level equality operator (Set)
  - *etc.*

Classifying the Collections (2)

- The classification by *external* functionality is not the only concern!
- The *internal* implementation, gives us additional dimensions
  - Array based implementation
    - Storing the elements in an array
  - Linked implementation
    - Linking the elements together
    - Several variations (*e.g.*, SkipList)
  - *etc.*

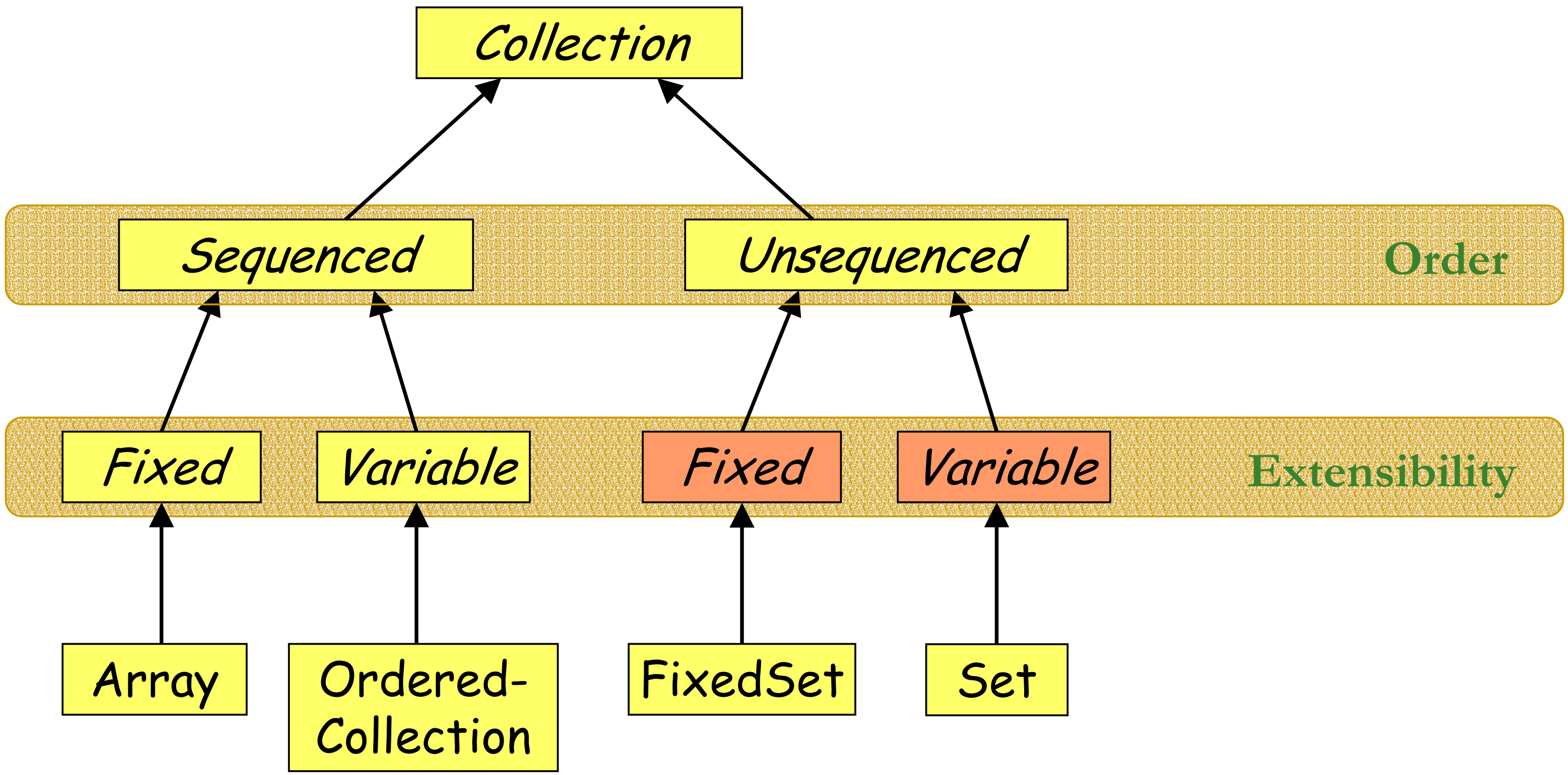
Implementation

- 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
- But: There are still many problems!

Implementation: The Problem

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

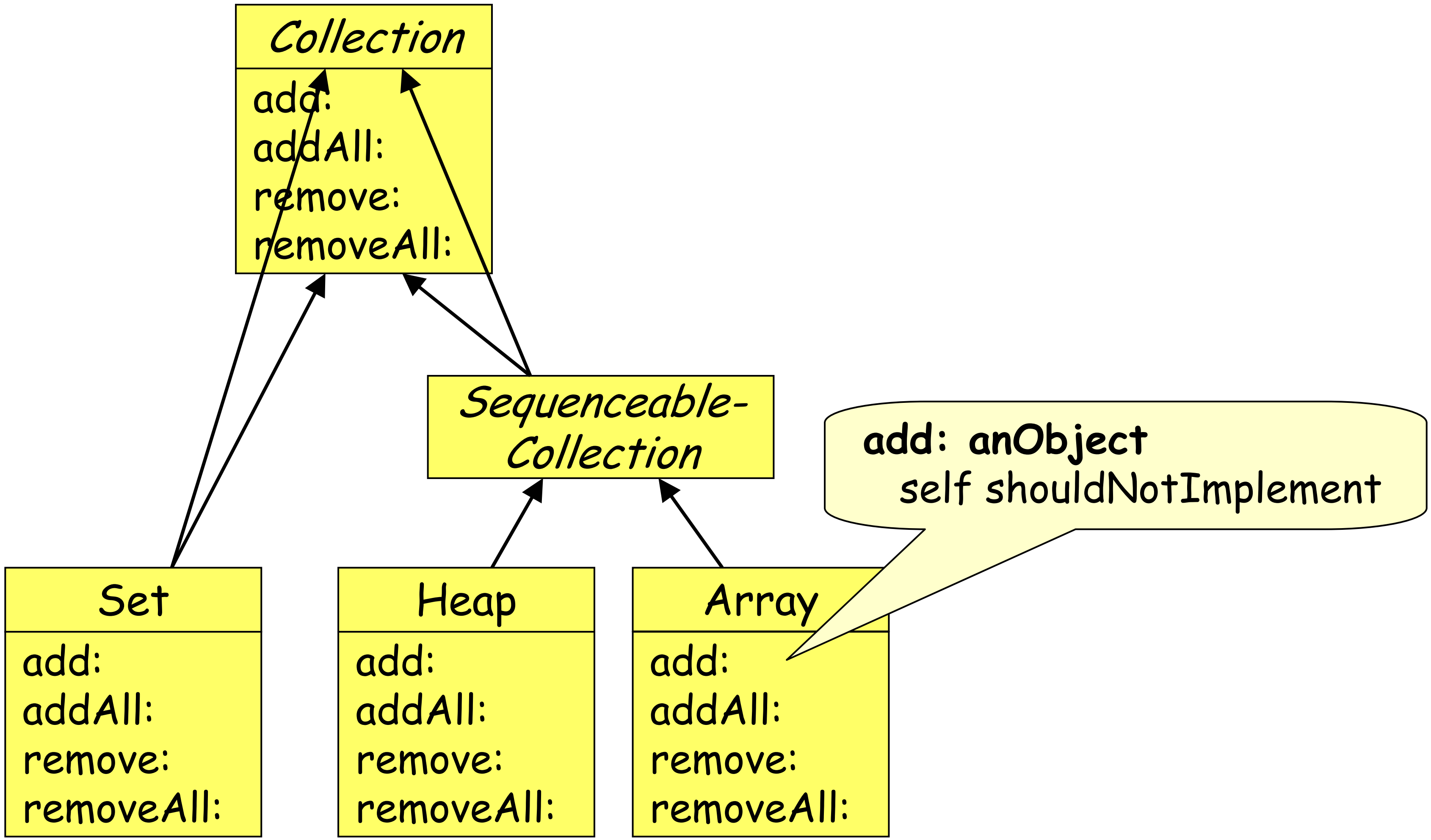
Implementation: The Problem (2)



Consequences

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

Methods implemented "too high"

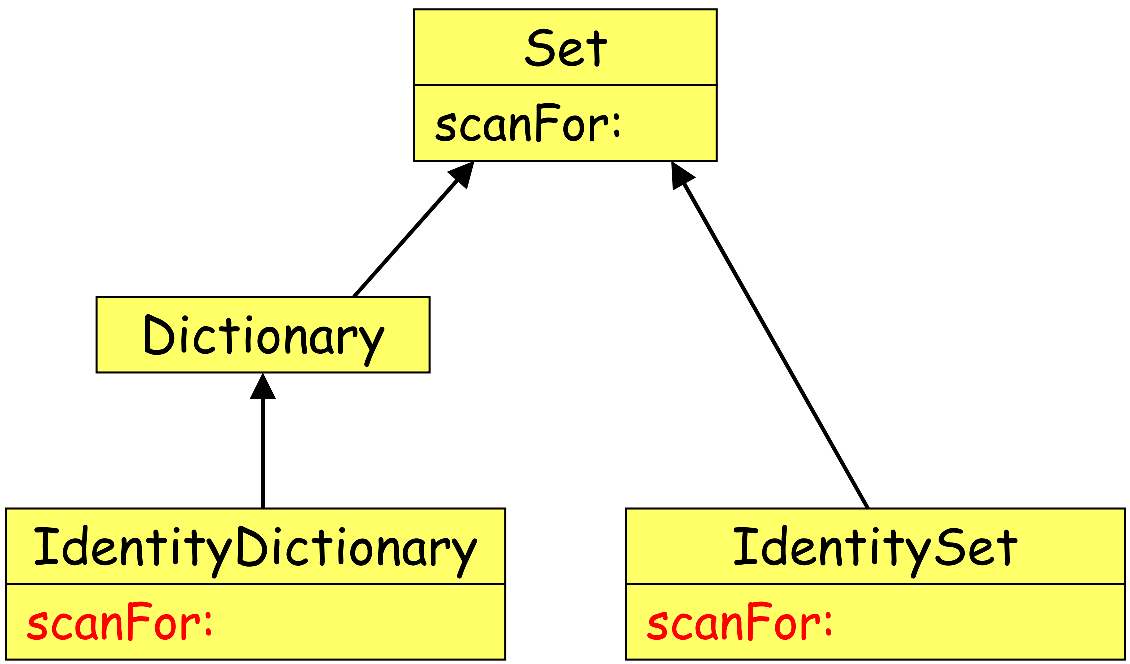


Methods implemented "too high" (2)

- Facts
  - 131 methods (more than 10%) are implemented too high
  - 106 methods (80%) are not explicitly "disabled"
- Consequences
  - Very hard to see what is the *real* interface of a class
  - Unexpected runtime errors

Code Duplication

- Happens when defining methods "too high" is not possible



Unnecessary Inheritance

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

Conceptual Shortcomings

- For `Sets` and `Dictionaries` there are different classes that exhibit different ways of comparing elements
  - `Set` (uses method `=`)
  - `IdentitySet` (uses method `==`)
  - `PluggableSet` (uses "pluggable comparsion block")
- But:
  - Corresponding variants are not available for other classes such as `Bag`, `Heap`, etc.
  - Implementing them is cumbersome because there is no way of factoring out and reusing these properties

Conceptual Shortcomings (2)

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

Roadmap

- Traits

Motivation

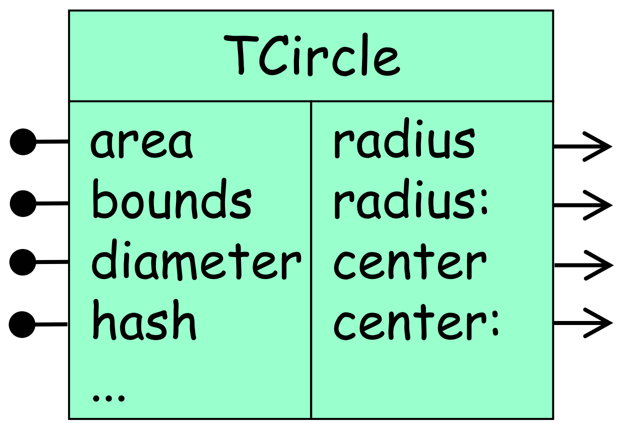
- The Smalltalk collection hierarchy shows:
  - Single Inheritance is inappropriate for many programming tasks
  - Even well-designed hierarchies suffer from the these deficiencies
- Nevertheless:
  - More powerful approaches such as multiple inheritance and mixins are not popular
  - Why?

Motivation (2)

- Multiple Inheritance
  - Hard to understand
  - Fragile with respect to changes
  - Limited compositional power
- Mixins
  - Surprising behavior
  - Dispersal of glue code
  - Fragile hierarchies
- Goal for Traits
  - Improving code reuse while avoiding these problems

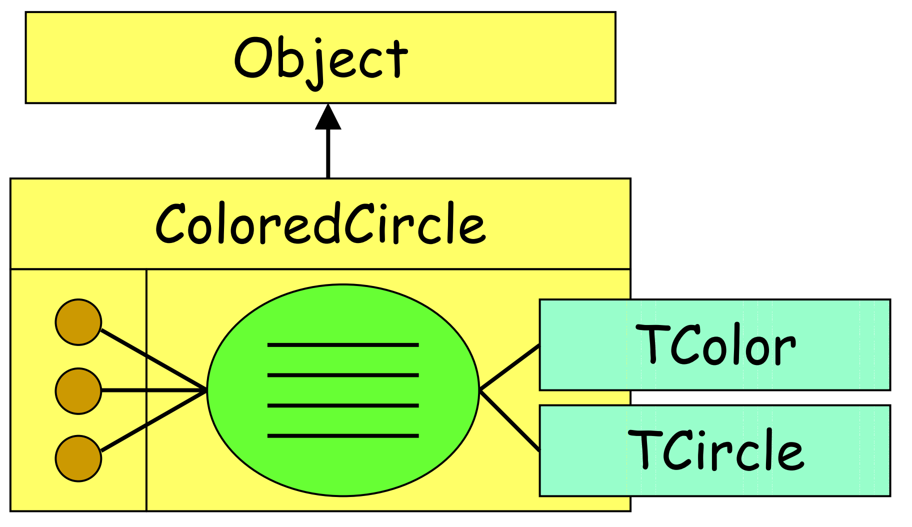
What are Traits?

- Traits are parameterized behaviors
  - Traits provide a set of methods (•—)
  - Traits require a set of methods (→)
  - Traits are purely behavioral
    - Traits do not specify any state



How are Traits Used?

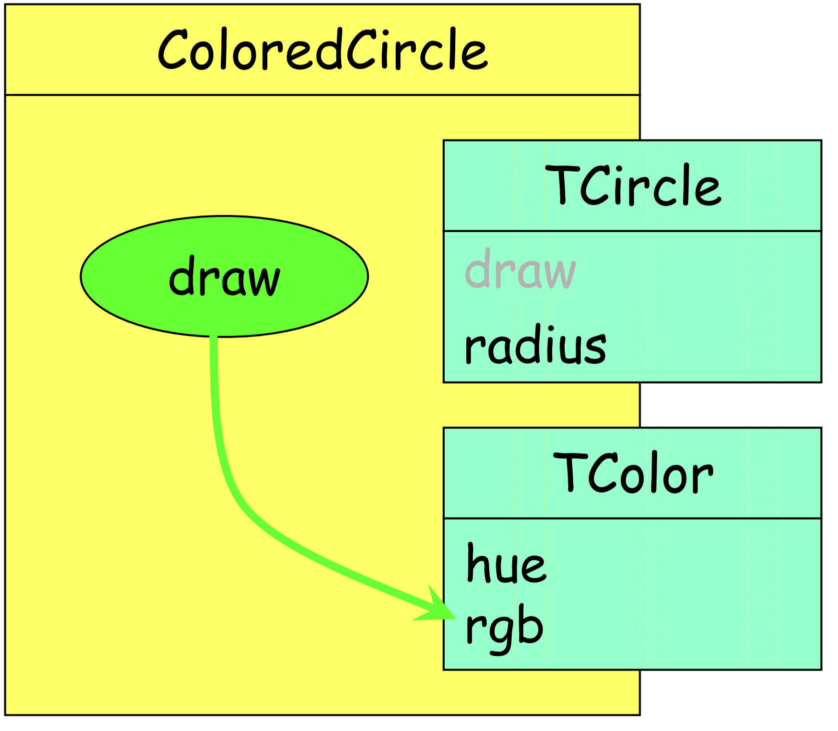
- Traits are the behavioral building blocks of classes
  - *Class = Superclass + State + Traits + Glue methods*



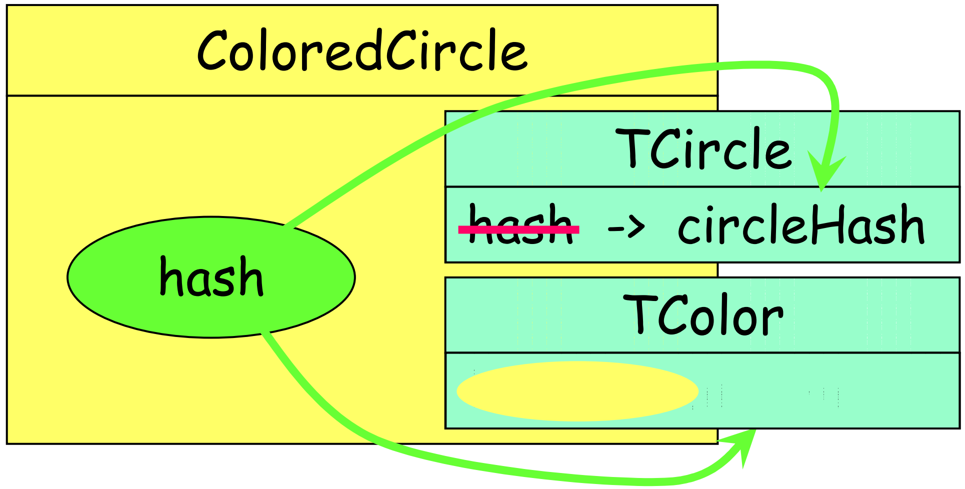
- Traits do not replace single inheritance
  - Instead, Traits provide modularity *within* classes

Composition Rules

- Class methods take precedence over trait methods



Conflicts Must be Resolved *Explicitly*



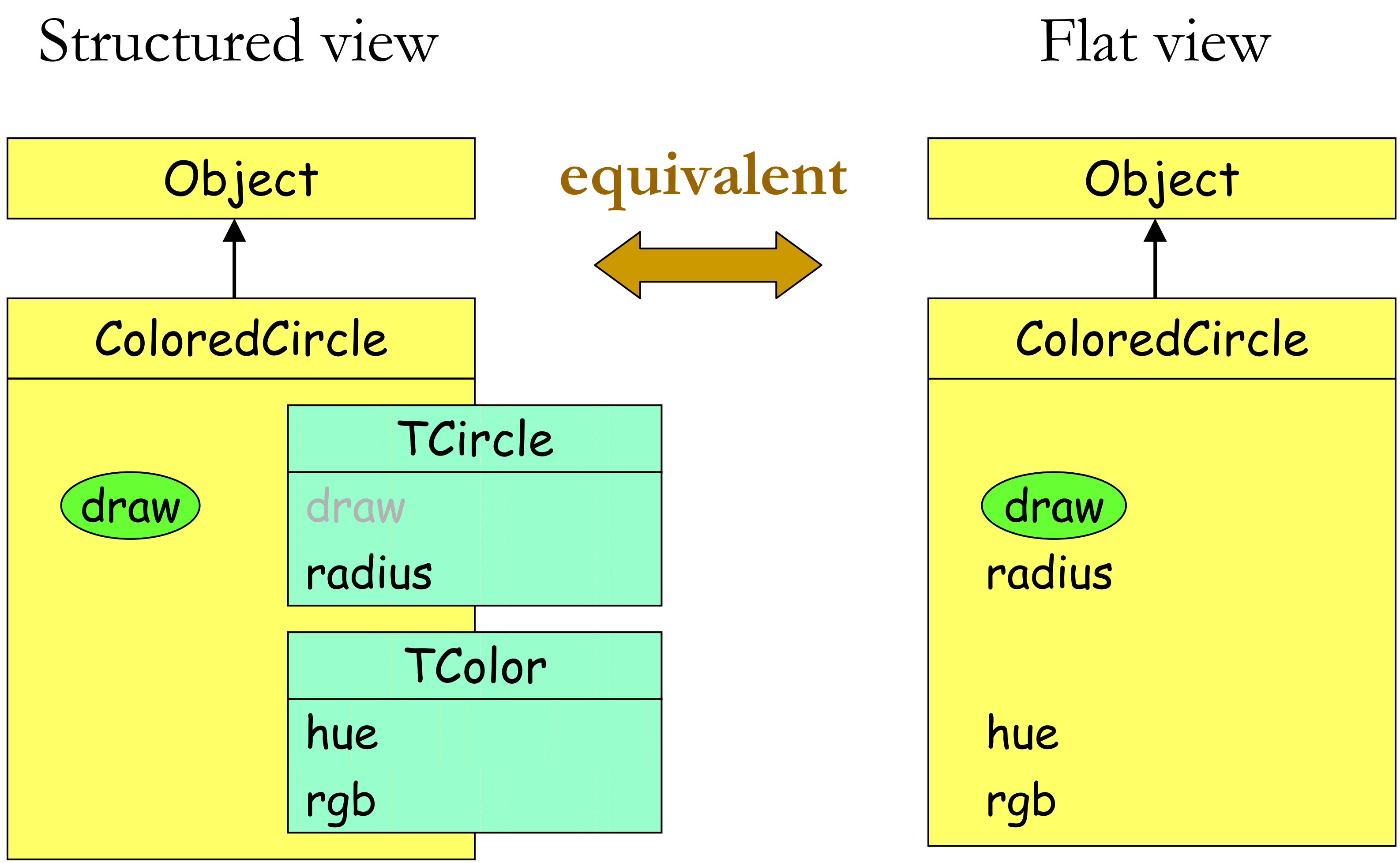
I) Override the conflict with a glue method

- Aliases provide access to the conflicting methods

II) Avoid the conflict

- Exclude the conflicting method from one trait

Flattening Property (Two views on code)



Roadmap

- Using Traits to Refactor the Collection Classes
  - Strategy and Process
  - Results and Discussion

Refactoring Strategy

- Creating 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 trait for one of its "sub-aspects" (*e.g.*, TEnumeration)
  - Make the class use this new trait and move the corresponding methods into the trait
  - Iterate...

Refactoring Process (2)

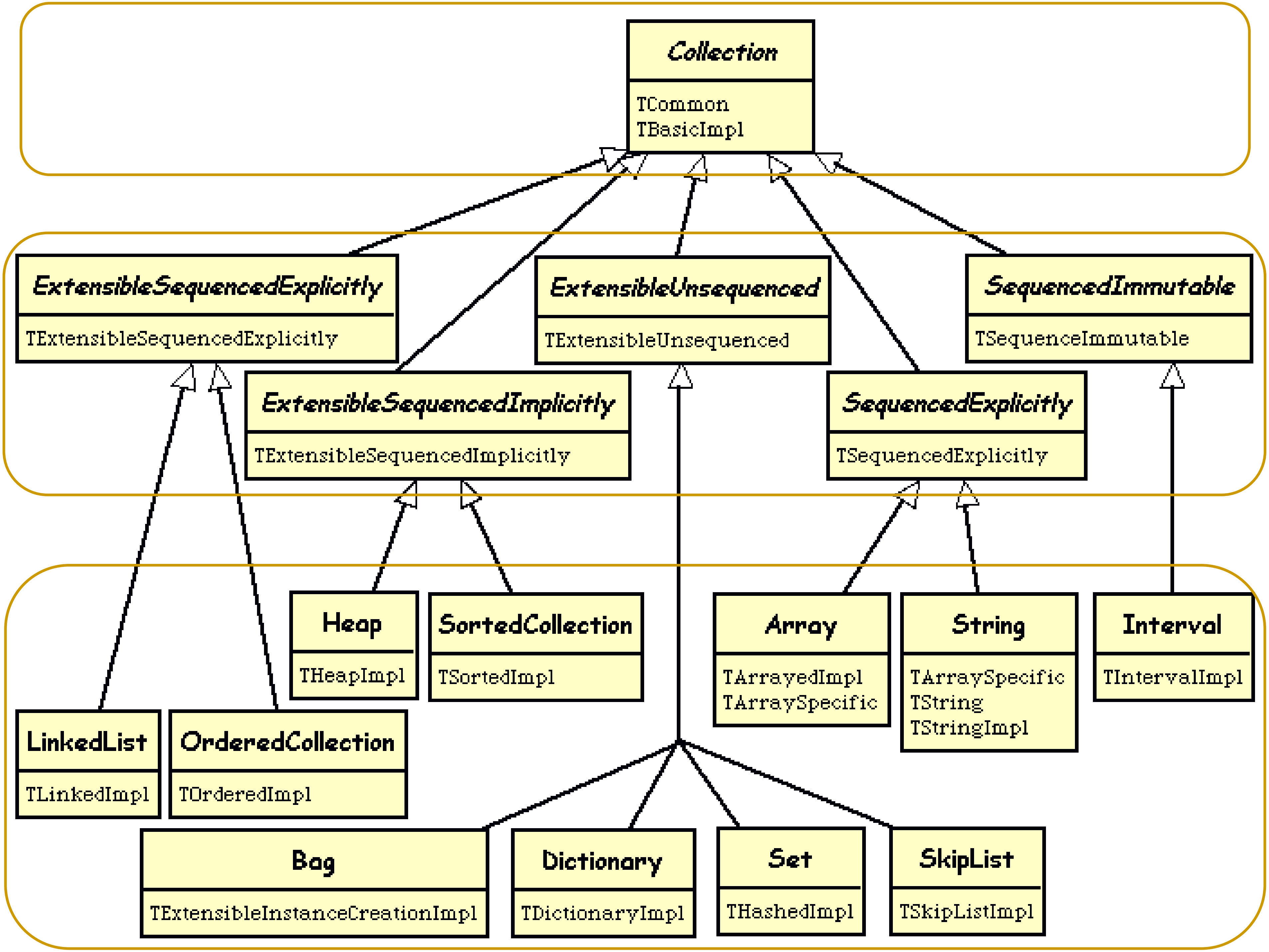
- Refinement steps:
  - Making the traits more fine-grained
    - Breaking traits into multiple subtraits
  - Juggling with classes and traits
    - Renaming classes and traits
    - Introducing new abstract classes
    - Replacing inheritance with trait composition

Refactoring Process (3)

- Live illustration!

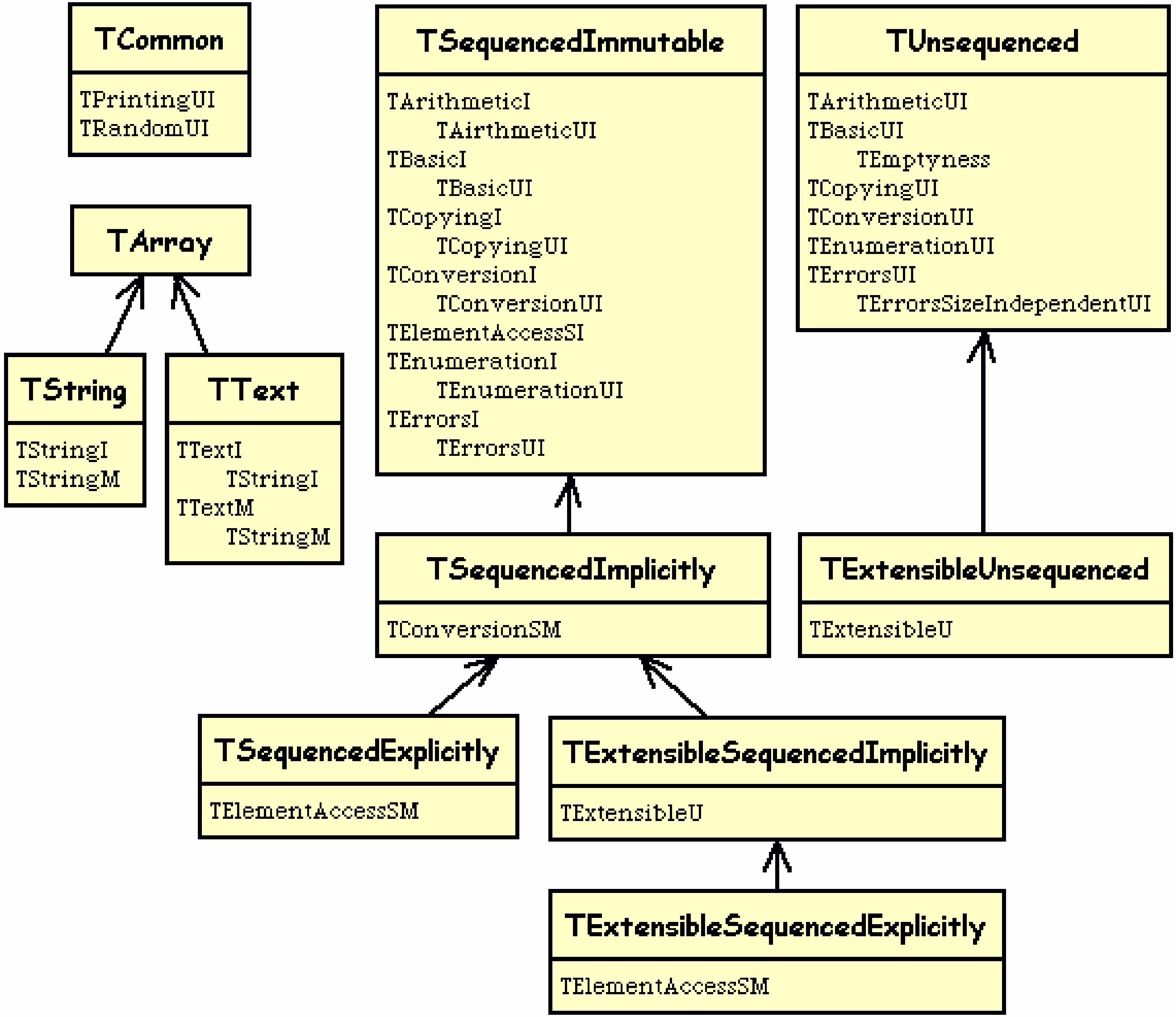
Resulting Class Hierarchy

- Resulting class hierarchy consists of 3 parts
  - Abstract root class **Collection**
    - Only contains the methods supported by *all* collection classes
  - Layer of abstract classes providing the public *functionality* of the different types of collections
  - Layer of concrete collection classes
    - They inherit the public functionality from one of the functionality classes
    - They use a trait that adds a specific *implementation*


Collection
TCommon
TBasicImpl
ExtensibleSequencedExplicitly
TExtensibleSequencedExplicitly
ExtensibleUnsequenced
TExtensibleUnsequenced
SequencedImmutable
TSequenceImmutable
ExtensibleSequencedImplicitly
TExtensibleSequencedImplicitly
SequencedExplicitly
TSequencedExplicitly
Heap
THeapImpl
SortedCollection
TSortedImpl
Array
TArrayedImpl
TArraySpecific
String
TArraySpecific
TString
TStringImpl
Interval
TIntervalImpl
LinkedList
TLinkedImpl
OrderedCollection
TOrderedImpl
Bag
TExtensibleInstanceCreationImpl
Dictionary
TDictionaryImpl
Set
THashedImpl
SkipList
TSkipListImpl

Resulting Trait Hierarchies

- Two trait hierarchies
  - Functional Traits
  - Implementation Traits
- Very fine-grained
  - Most traits consist of multiple subtraits


TCommon
TPrintingUI
TRandomUI
TArray
TString
TStringI
TStringM
TText
TTextI
TStringI
TTextM
TStringM
TSequencedImmutable
TArithmeticI
TAirthmeticUI
TBasicI
TBasicUI
TCopyingI
TCopyingUI
TConversionI
TConversionUI
TElementAccessSI
TEnumerationI
TEnumerationUI
TErrorsI
TErrorsUI
TUnsequenced
TArithmeticUI
TBasicUI
TEmptyness
TCopyingUI
TConversionUI
TEnumerationUI
TErrorsUI
TErrorsSizeIndependentUI
TSequencedImplicitly
TConversionSM
TExtensibleUnsequenced
TExtensibleU
TSequencedExplicitly
TElementAccessSM
TExtensibleSequencedImplicitly
TExtensibleU
TExtensibleSequencedExplicitly
TElementAccessSM

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?

- Consistent hierarchies
  - 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 are implemented "too high"
    - No "disabling" of inherited methods necessary
  - No missing methods
    - Classes satisfy all the appropriate protocols
    - Collections can be used more uniformly

What did we gain? (2)

- Less code!
  - Elimination of code duplication
  - About 10% less source code
  - More than 20% fewer methods

What did we gain? (3)

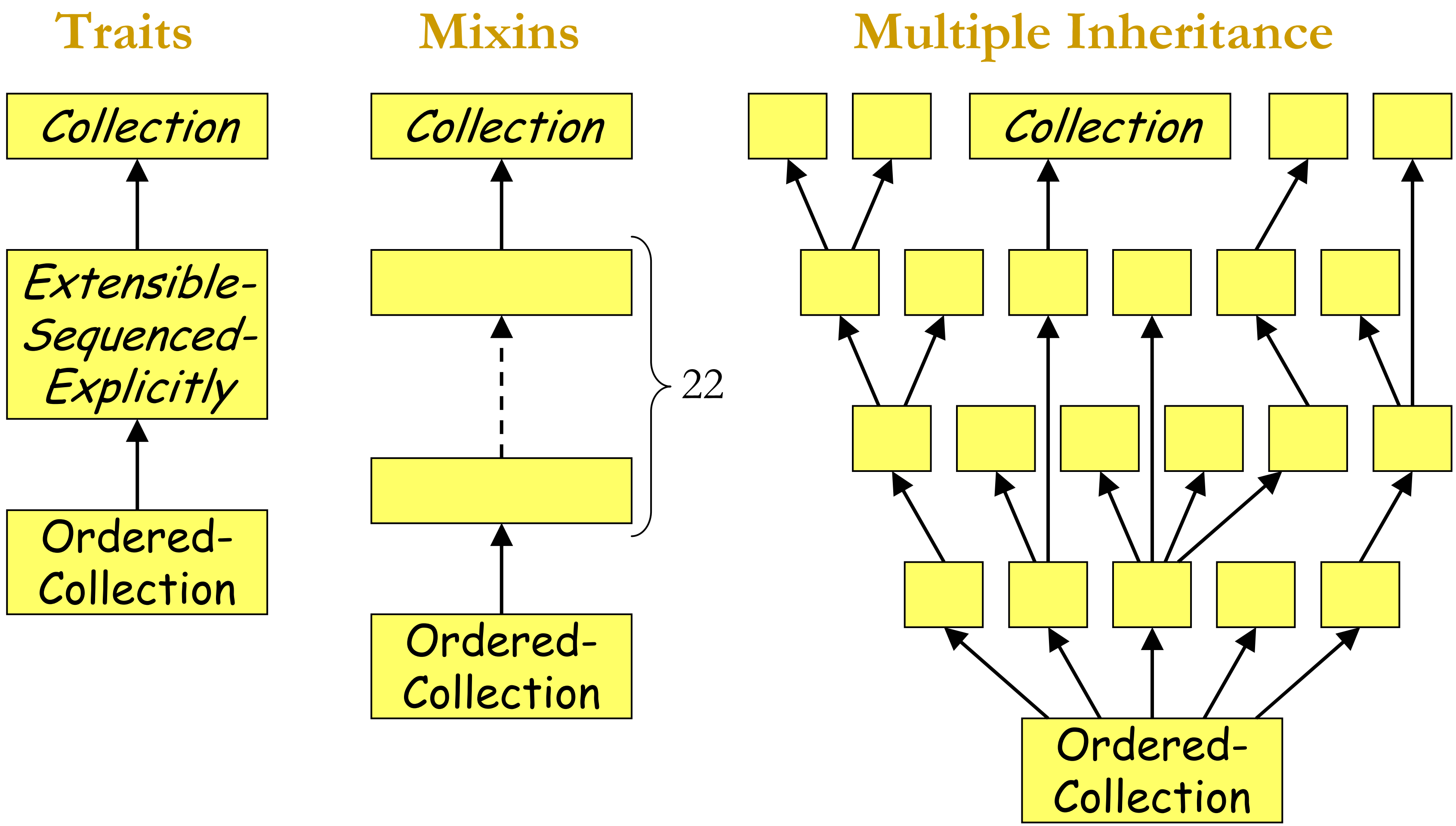
- Improved reusability
  - The "toolbox of traits" makes it much easier to create new collection classes
    - *E.g.*, a class `PluggableBag` can be created as a subclass of `Bag` by using the trait `TPluggableAdaptor`
  - Traits can be reused outside of the collection hierarchy
    - *E.g.*, the trait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

```
newPath := path collect:
                [:each | rect containsPoint: each]
```

Why Traits (vs. Mixins and MI)?

- Main reason: Flattening property
  - Although classes are built from up to 22 traits, we can still *view and work* with these classes in the traditional (*i.e.*, single inheritance) way
  - There is no trade-off between finer granularity and understandability
    - We introduced more traits than we would have needed to improve understandability and unanticipated reuse
- This does not hold for mixins and multiple inheritance

Why Traits (vs. Mixins and MI)?? (2)



Roadmap

- Related Work and Conclusion

Related Work

- 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 ST (using mixins)
  - Both approaches are much less fine-grained
- A lot of work on refactoring
  - Automated refactoring

Conclusion

- Theoretical properties of traits paid off in practice
  - Flattening property
  - Sum operation with explicit conflicts
- Migration 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 have no disadvantages
- Refactoring with traits was fun!
  - Tools are important