Refactoring and Lattice Theory

Jeffrey Clark

Elon University
clarkj@elon.edu

March 16, 2007

Introduction

- Refactoring refers to revising the implementation of objects in object-oriented programming while maintaining their behavior.

Introduction

- Refactoring refers to revising the implementation of objects in object-oriented programming while maintaining their behavior.
- A common simplifying refactoring is to replace a group of attributes by a single attribute.

Introduction

- Refactoring refers to revising the implementation of objects in object-oriented programming while maintaining their behavior.
- A common simplifying refactoring is to replace a group of attributes by a single attribute.
- It is possible to detect when this is possible by using lattice theory.

Availability of Presentation

This talk is available from my web-site:

`http://frodo.elon.edu`

under the link Presentations.

Classes, Attributes, and Methods

- A class is a template for creating an object, which is a way of organizing data and behavior for the data.

Classes, Attributes, and Methods

- A class is a template for creating an object, which is a way of organizing data and behavior for the data.
- A class can have attributes, which every instantiation (object) of that class will have. The attributes are where the data is stored.

Classes, Attributes, and Methods

- A class is a template for creating an object, which is a way of organizing data and behavior for the data.
- A class can have attributes, which every instantiation (object) of that class will have. The attributes are where the data is stored.
- The behavior of the class is defined in its methods, which are the code used to access and modify the data in the attributes.

Example: PermutationWord Class

- PermutationWord class serves as template for creating permutation objects that keep track of their expression in terms of group generators

Example: PermutationWord Class

- PermutationWord class serves as template for creating permutation objects that keep track of their expression in terms of group generators
- Attribute: degree

Example: PermutationWord Class

- PermutationWord class serves as template for creating permutation objects that keep track of their expression in terms of group generators
- Attribute: degree
- Attribute: image (image of permutation applied to $[1, \ldots, \text{degree}]$

Example: PermutationWord Class

- PermutationWord class serves as template for creating permutation objects that keep track of their expression in terms of group generators
- Attribute: degree
- Attribute: image (image of permutation applied to $[1, \ldots, \text{degree}]$
- Attribute: word (representation in terms of group generators)

Example: PermutationWord Class

- PermutationWord class serves as template for creating permutation objects that keep track of their expression in terms of group generators
- Attribute: degree
- Attribute: image (image of permutation applied to $[1, \ldots, \text{degree}]$
- Attribute: word (representation in terms of group generators)
- Method: compositionWith

Example: PermutationWord Class

- PermutationWord class serves as template for creating permutation objects that keep track of their expression in terms of group generators
- Attribute: degree
- Attribute: image (image of permutation applied to $[1, \ldots, \text{degree}]$
- Attribute: word (representation in terms of group generators)
- Method: compositionWith
- Method: getDegree

Example: PermutationWord Class

- PermutationWord class serves as template for creating permutation objects that keep track of their expression in terms of group generators
- Attribute: degree
- Attribute: image (image of permutation applied to $[1, \ldots, \text{degree}]$
- Attribute: word (representation in terms of group generators)
- Method: compositionWith
- Method: getDegree
- Method: getInverse

Example: PermutationWord Class

- PermutationWord class serves as template for creating permutation objects that keep track of their expression in terms of group generators
- Attribute: degree
- Attribute: image (image of permutation applied to $[1, \ldots, \text{degree}]$
- Attribute: word (representation in terms of group generators)
- Method: compositionWith
- Method: getDegree
- Method: getInverse
- Method: getOrder

Example: PermutationWord Class

- PermutationWord class serves as template for creating permutation objects that keep track of their expression in terms of group generators
- Attribute: degree
- Attribute: image (image of permutation applied to $[1, \ldots, \text{degree}]$
- Attribute: word (representation in terms of group generators)
- Method: compositionWith
- Method: getDegree
- Method: getInverse
- Method: getOrder
- Method: getString

Example: PermutationWord Class

- PermutationWord class serves as template for creating permutation objects that keep track of their expression in terms of group generators
- Attribute: degree
- Attribute: image (image of permutation applied to $[1, \ldots, \text{degree}]$
- Attribute: word (representation in terms of group generators)
- Method: compositionWith
- Method: getDegree
- Method: getInverse
- Method: getOrder
- Method: getString
- Method: getWord

Method Attribute Signatures

- Each method accesses or modifies certain attributes. The attributes that a method depends on are called its attribute signature.

Method Attribute Signatures

- Each method accesses or modifies certain attributes. The attributes that a method depends on are called its attribute signature.
- In the PermutationWord class, the methods have the following attribute signatures:

Method Attribute Signatures

- Each method accesses or modifies certain attributes. The attributes that a method depends on are called its attribute signature.
- In the PermutationWord class, the methods have the following attribute signatures:

  compositionWith: degree, image, word

Method Attribute Signatures

- Each method accesses or modifies certain attributes. The attributes that a method depends on are called its attribute signature.
- In the PermutationWord class, the methods have the following attribute signatures:

  compositionWith: degree, image, word
  getDegree: degree

Method Attribute Signatures

- Each method accesses or modifies certain attributes. The attributes that a method depends on are called its attribute signature.
- In the PermutationWord class, the methods have the following attribute signatures:

  compositionWith: degree, image, word
  getDegree: degree
  getInverse: degree, image, word

Method Attribute Signatures

- Each method accesses or modifies certain attributes. The attributes that a method depends on are called its attribute signature.
- In the PermutationWord class, the methods have the following attribute signatures:

  compositionWith: degree, image, word
  getDegree: degree
  getInverse: degree, image, word
  getOrder: image

Method Attribute Signatures

- Each method accesses or modifies certain attributes. The attributes that a method depends on are called its attribute signature.
- In the PermutationWord class, the methods have the following attribute signatures:

  compositionWith: degree, image, word
  getDegree: degree
  getInverse: degree, image, word
  getOrder: image
  getString: degree, image

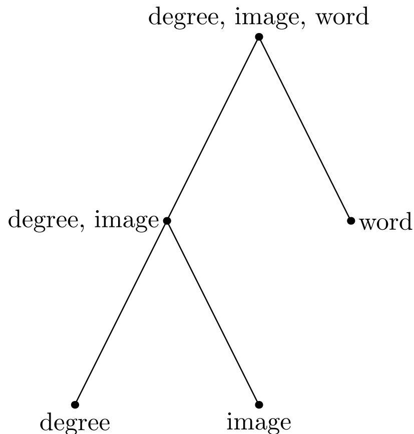
Method Attribute Signatures

- Each method accesses or modifies certain attributes. The attributes that a method depends on are called its attribute signature.
- In the PermutationWord class, the methods have the following attribute signatures:

  compositionWith: degree, image, word
  getDegree: degree
  getInverse: degree, image, word
  getOrder: image
  getString: degree, image
  getWord: word

Lattice of Signatures

The sets of attributes in the signatures form a lattice.



Ideals

- This lattice for PermutationWord has an ideal located at and below the degree and image attributes.

Ideals

- This lattice for PermutationWord has an ideal located at and below the degree and image attributes.
- In lattice theory, an ideal is a sub-lattice I such that:

Ideals

- This lattice for PermutationWord has an ideal located at and below the degree and image attributes.
- In lattice theory, an ideal is a sub-lattice I such that:
  - The join of any two elements of I is in I.

Ideals

- This lattice for PermutationWord has an ideal located at and below the degree and image attributes.
- In lattice theory, an ideal is a sub-lattice I such that:
  - The join of any two elements of I is in I.
  - Any element $x \leq y$ for y in I is also in I.

Ideals

- This lattice for PermutationWord has an ideal located at and below the degree and image attributes.
- In lattice theory, an ideal is a sub-lattice I such that:
  - The join of any two elements of I is in I.
  - Any element $x \leq y$ for y in I is also in I.
- The signatures at and below “degree, image” in the lattice form an ideal.

Ideals

- This lattice for PermutationWord has an ideal located at and below the degree and image attributes.
- In lattice theory, an ideal is a sub-lattice I such that:
  - The join of any two elements of I is in I.
  - Any element $x \leq y$ for y in I is also in I.
- The signatures at and below "degree, image" in the lattice form an ideal.
- This is also an example of a principal ideal, namely one that is determined by an upper bound (in this case the degree and image attributes).

Ideals

- This lattice for PermutationWord has an ideal located at and below the degree and image attributes.
- In lattice theory, an ideal is a sub-lattice I such that:
  - The join of any two elements of I is in I.
  - Any element $x \leq y$ for y in I is also in I.
- The signatures at and below "degree, image" in the lattice form an ideal.
- This is also an example of a principal ideal, namely one that is determined by an upper bound (in this case the degree and image attributes).
- In a finite lattice, every ideal is a principal ideal.

Extracting Classes

- One of the most common refactorings is to extract a class out of a subset of attributes in a given class.

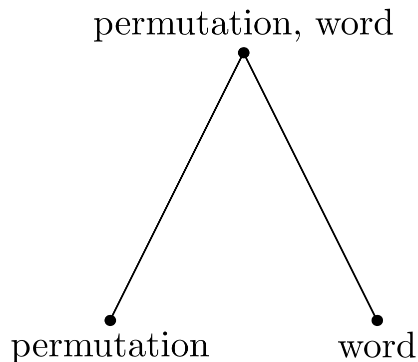
Extracting Classes

- One of the most common refactorings is to extract a class out of a subset of attributes in a given class.
- It is awkward to extract out a class if the attributes if there are methods that include some of the attributes but not others.

Extracting Classes

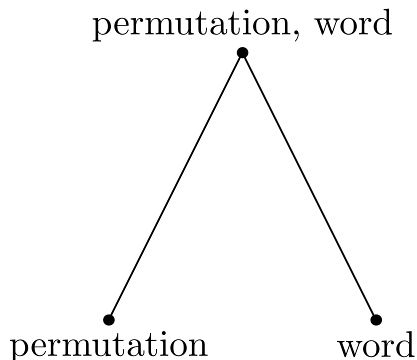
- One of the most common refactorings is to extract a class out of a subset of attributes in a given class.
- It is awkward to extract out a class if the attributes if there are methods that include some of the attributes but not others.
- When you extract a set of attributes that form a principal ideal in the attributes lattice, all of the methods are either part of the new class are separate from it.

Refactored Example



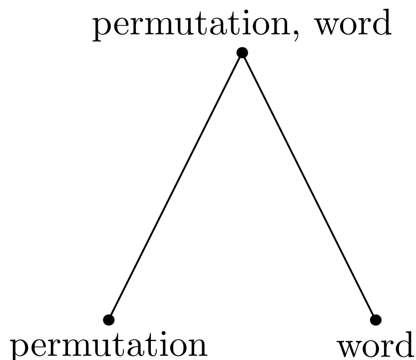
- Extract degree, image attributes into new Permutation class.

Refactored Example



- Extract degree, image attributes into new Permutation class.
- PermutationWord method signatures become:

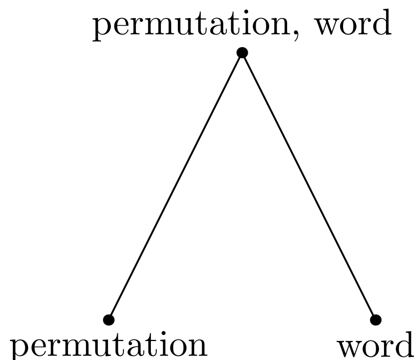
Refactored Example



- Extract degree, image attributes into new Permutation class.
- PermutationWord method signatures become:

  compositionWith:
  permutation,
  word

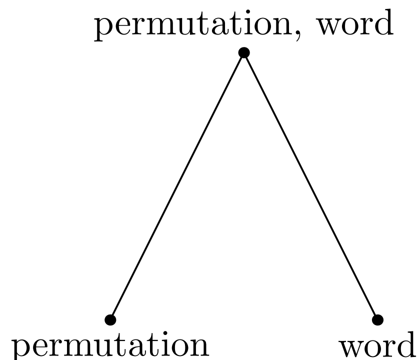
Refactored Example



- Extract degree, image attributes into new Permutation class.
- PermutationWord method signatures become:

  compositionWith:
  permutation, word
  getInverse: permutation, word

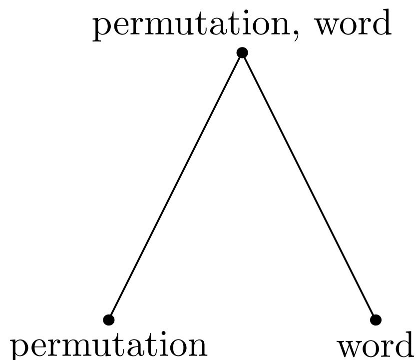
Refactored Example



- Extract degree, image attributes into new Permutation class.
- PermutationWord method signatures become:

compositionWith:	permutation, word
getInverse:	permutation, word
getString:	permutation

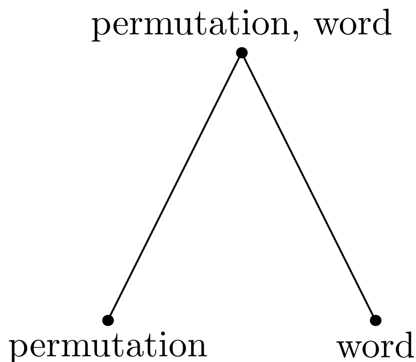
Refactored Example



- Extract degree, image attributes into new Permutation class.
- PermutationWord method signatures become:

compositionWith:	permutation, word
getInverse:	permutation, word
getString:	permutation
getWord:	word

Refactored Example

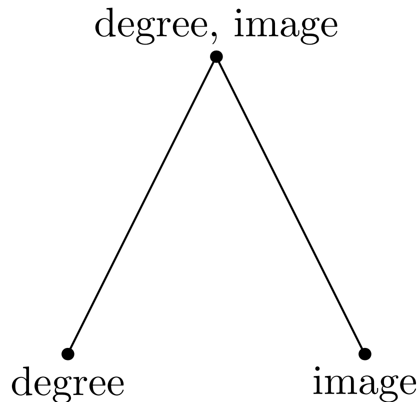


- Extract degree, image attributes into new Permutation class.
- PermutationWord method signatures become:

compositionWith:	permutation, word
getInverse:	permutation, word
getString:	permutation
getWord:	word

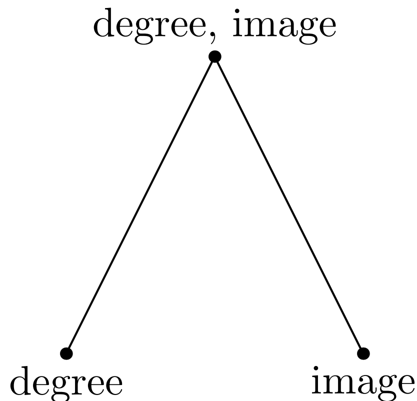
- If any of the deleted methods need to be called publicly, we can delegate them to permutation.

Refactored Example (Continued)



- Permutation method signatures become:

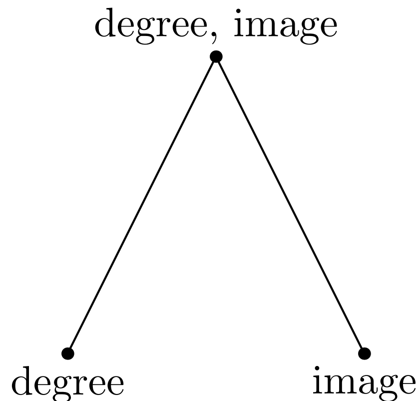
Refactored Example (Continued)



- Permutation method signatures become:

  getDegree: degree

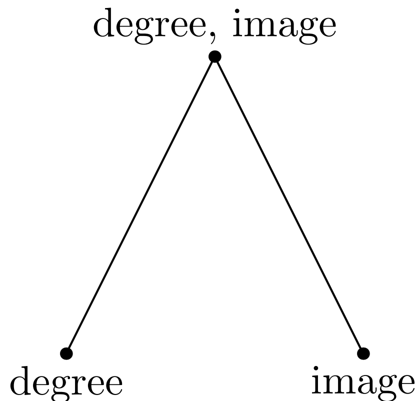
Refactored Example (Continued)



- Permutation method signatures become:

getDegree:	degree
getOrder:	image

Refactored Example (Continued)



- Permutation method signatures become:

getDegree:	degree
getOrder:	image
getString:	degree, image

Conclusion

- Classes with many attributes and many methods are hard to validate and to maintain.

Conclusion

- Classes with many attributes and many methods are hard to validate and to maintain.
- A common refactoring is to identify a subset of attributes that have their own behavior, and to extract them into their own class.

Conclusion

- Classes with many attributes and many methods are hard to validate and to maintain.
- A common refactoring is to identify a subset of attributes that have their own behavior, and to extract them into their own class.
- A natural way to identify which attributes to extract is to look at attribute signatures for the methods, and to look for any non-trivial principal ideals.

Conclusion

- Classes with many attributes and many methods are hard to validate and to maintain.
- A common refactoring is to identify a subset of attributes that have their own behavior, and to extract them into their own class.
- A natural way to identify which attributes to extract is to look at attribute signatures for the methods, and to look for any non-trivial principal ideals.
- A non-trivial principal ideal indicates a natural candidate for extraction of the respective attributes.