Full Application of the Extract Interface Refactoring: Conceptual Structures in the Hands of Master Students

Marianne Huchard

Full Application of the Extract Interface Refactoring

Conceptual Structures in the Hands of Master Students

Marianne Huchard
LIRMM, CNRS and Montpellier University
Montpellier, France
huchard@lirmm.fr

ABSTRACT

Interfaces are data types that are very useful for providing abstract and organized views on programs and APIs, and opportunities for writing more generic code and for reuse. Extract interface refactoring is a well known local refactoring which is commonly used in development tools. Beyond that local refactoring, there is a need for mass extraction of an interface hierarchy from a class hierarchy. In this paper, we made an experience with master students to put into practice an existing Formal Concept Analysis (FCA) based approach for solving that problem. The results show that the data selection (selected datatypes: interfaces, abstract classes, concrete classes; attributes; attribute description; methods; method description; etc.) was not obvious as it was expected to be, and that the students used the approach more as an analysis technique that would guide the extraction, than as a turn key solution.

CCS Concepts

•Software and its engineering → Software reverse engineering; •Mathematics of computing → *Graph theory;*

Keywords

Extract interface refactoring, Java collection API, Formal Concept analysis

1. INTRODUCTION

Offering abstract views of programs and APIs is a fundamental activity in forward as well as in reverse software development. These abstract views can be used for understanding purposes, to promote reusability or to facilitate maintenance and evolution. Different techniques are used, among which interfaces play a central role.

- Highlighting interfaces allows developers to separate more clearly specification from implementation.
- Classes often group several responsibilities, that should be separated in several interfaces. Once highlighted, the responsibility subsets may correspond to roles and serve as simplistic contracts for a class.
- The interface specialization hierarchy offers a conceptual classification, closer to the represented domain entities (or data types) than a class hierarchy whose design has been partly guided by technical concerns.
- The existence of the interfaces allows developers to write generic code, using only specified operations and avoiding the use of technical tricks.

For these reasons, the *extract interface refactoring (or EIR for short)* is one of the important refactorings proposed by Fowler et al. [1] for writing a good quality code and it is implemented in several IDEs such as the eclipse platform[1]. Beyond that local refactoring, other research investigated a *full application of extract interface refactoring* (FAEIR) in order to extract in a systematic and exhaustive way all the interfaces from a class hierarchy as a whole [3, 6]. The additional benefit of these approaches, that are based on Formal Concept Analysis [2], lays in gathering (in a single interface) extracted method signatures that otherwise would be duplicated several times, and in organizing the extracted interfaces in a single hierarchy with strong theoretical properties.

In Java, an *interface* is a specific artifact that groups public abstract methods, public static final fields, and since Java 1.8, default implementations of methods and static methods. The interface hierarchy offers multiple specialization (with `extends` keyword), while the class hierarchy uses single specialization (with the same `extends` keyword). A class can implement several interfaces (with `implements` keyword). This strengthens the capacity of the interface hierarchy to better represent the modeled domain.

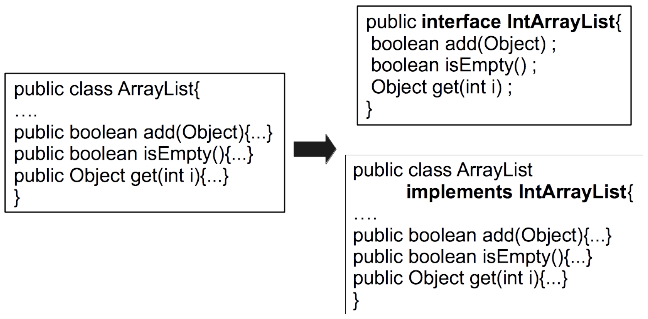
This paper reports a practical experience done with master students, a few months before the internship in software industry that concludes their studies. The students had to apply FAEIR on the Java 1.8 Collections and Maps API using the method presented in [3, 6]. Initially the objective was to check if this complex refactoring was easy to put into practice by the students and the feasibility of adding it as a component in IDE refactoring suggestions (and how). The experience showed unexpected difficulties from which we can define guidelines for using FAEIR.

[1]http://help.eclipse.org/luna/topic/org.eclipse.jdt.doc.user/concepts/concept-refactoring.htm

Section 2 presents the local and the full refactorings on examples and outlines the method used for the full application of extract interface refactoring. Then, Section 3 explains the conditions under which the experience has been done. The variations on the datasets used by the students are presented in Section 4. The way the students analyzed the results is reported in Section 5. Section 6 concludes the paper.

2. EXTRACT INTERFACE REFACTORING

Figure 1 shows how the Extract Interface Refactoring (EIR) works. In this case, the signatures of the public (non-static) methods, including name, parameter type list and return type of class `ArrayList` are extracted and inserted in interface `IntArrayList` which is created for this purpose. Then `ArrayList` is updated to declare that it implements `IntArrayList`.



```
public class ArrayList{
....
public boolean add(Object){...}
public boolean isEmpty(){...}
public Object get(int i){...}
}
```

```
public interface IntArrayList{
 boolean add(Object) ;
 boolean isEmpty() ;
 Object get(int i) ;
}
```

```
public class ArrayList
        implements IntArrayList{
....
public boolean add(Object){...}
public boolean isEmpty(){...}
public Object get(int i){...}
}
```

Figure 1: Extract Interface Refactoring (EIR) example: extraction of the ArrayList interface

Input data for full application of EIR based on Formal Concept Analysis is not a single class, but a set of classes. Figure 2(a) shows a set of four simplified sequence classes, given with the signatures of their public non-static methods. Applying EIR to each class would give four interfaces sharing some signatures, and with no specialization relations.

To extract at the same time and rightly organize the interfaces without signature duplication, the approaches in [3, 6] use a specific substructure of the concept lattice built on top of the binary relation (called a formal context in FCA terminology) of Figure 2(a). The concept lattice is formed by computing the formal concepts of the formal context. In our framework, a formal concept is a maximal set of classes sharing a maximal set of signatures. Formal concepts are ordered in the lattice using inclusion of their class set (or equivalently of their signature set).

Lattices are represented by their Hasse diagram (Figure 2(b)), with upward paths going from more specific to more general concepts. Concepts are represented in 3-part boxes containing: the concept name, the intent (shared signatures), the extent (corresponding classes). They are shown with simplified labeling where a signature is shown only in the highest concept that introduces it (and inherited by the subconcepts), while a class is shown only in the lowest concept that introduces it (and inherited by the superconcepts). *Concept_SimpSeq_2* has thus its intent composed of `add, isEmpty` (both top-to-bottom inherited), and `peek, poll` (both introduced locally) and its extent composed of `PrioriyQueue, ArrayDeque` (both introduced locally), and `LinkedList` (inherited bottom-to-top).

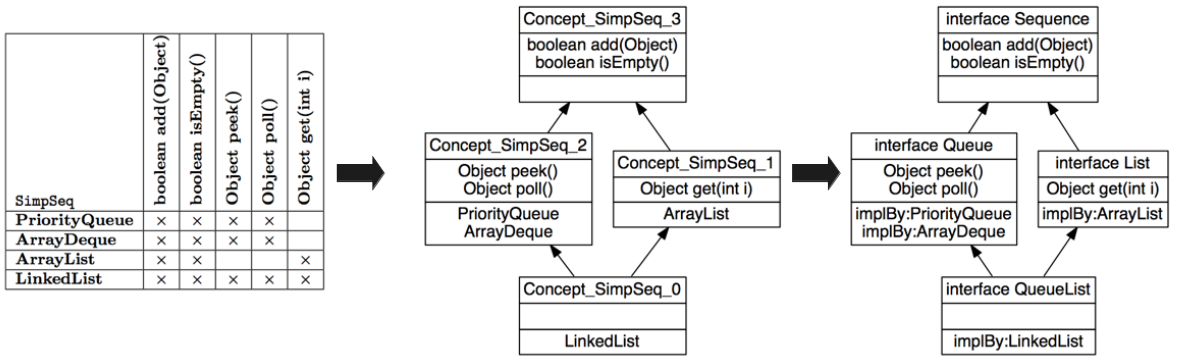
Extracting the interface hierarchy is easy, and shown in Figure 2(c). Each concept is interpreted as an interface, the intent gives the set of operations of the interface and we also show which class will directly implement which interface. The resulting hierarchy has strong theoretical properties: it is the most compact (in number of interfaces) hierarchy that avoids any signature duplication and it contains all the specialization links (if an interface $Isub$ contains all the signatures of an interface $Isup$, $Isub$ necessarily specializes $Isup$ in the hierarchy). Notice that, due to the compactness property, the resulting structure may group signatures that correspond to different roles if these roles are always present together in implementing classes. Thus it may need to be reworked, in order to divide the signatures declared in a concept into subsets corresponding to roles. In all known usages of that technique, the AOC-poset is used rather the concept lattice. The AOC-poset is restricted to concepts that introduce at least one signature or at least one class. Following its definition, its size ($\#concepts$) is bounded by ($\#classes + \#signatures$), which may be large, but very reasonable compared to the whole concept lattice whose size may get $2^{min(\#classes,\#signatures)}$. Some students used iceberg lattices. These structures have been proposed by [8] for analyzing very large databases and visualizing association rules. An iceberg lattice is restricted to the concepts which have an extent containing more than a certain proportion of the initial class set.

As noticed in [1], this refactoring has similarities with Extract Superclass refactoring which remains a current concern [7]. This is why, similar techniques have also been developed to apply full extract superclass refactoring [4, 5].

Figure 3 shows the AOC-poset extracted from a subset of six main Java 1.8 concrete classes representing sequences: `ArrayList`, `Vector`, `Stack`, `PriorityQueue`, `ArrayDeque` and `LinkedList`. All public signatures (including return type and parameter type list) have been included, giving an overview of a real case. The methods of the Java `Object` class have been removed. They are not relevant here because they appear in all classes (they are not specifically common to sequences).

All the concepts may be used as interfaces. Several of them are detailed to illustrate information contained in this structure. In Fig. 3, the top concept `Concept_Coll_12` highlights what is common to all classes. It can be interpreted as a "Sequence" interface. `Concept_Coll_11` groups the sequences (all except `PriorityQueue`) that have a public `clone` method (it is protected in `Object`). It corresponds to a "CloneableSequence" interface. `Concept_Coll_7` can be called "Popable" interface. `Concept_Coll_6` introduced the methods common to `Vector` and `ArrayList`. As `Vector` is introduced in a subconcept of the concept which introduces `ArrayList`, this means that its interface includes that of `ArrayList`, with the additional operations that can be seen in `Concept_Coll_4`. The same situation occur for `Stack` which is introduced below `Vector`. But although `Stack` is a subclass of `Vector` in the Java collection API, `Vector` is a cousin class of `ArrayList`.

Here, the iceberg lattice for a chosen proportion of 50% of the 6 initial classes contains concepts `Concept_Coll_6`, `Concept_Coll_7`, `Concept_Coll_8` and their super-concepts (extents increase in bottom-up paths) because they contain 3 or more classes in their extent, but not their subconcepts which have less than 3 classes in their extent.



SimpSeq	boolean add(Object)	boolean isEmpty()	Object peek()	Object poll()	Object get(int i)
PriorityQueue	×	×	×	×	
ArrayDeque	×	×	×	×	
ArrayList	×	×			×
LinkedList	×	×	×	×	×

(a) Formal Context describing classes by signatures of public methods

(b) Concept lattice showing maximal factorization and specialization

(c) Interface hierarchy annotated by implementing classes

Figure 2: Full Application of EIR based on Formal Concept Analysis on 4 simplified classes

3. PREPARATION OF THE WORK

The practical work asked to the students consisted in *extracting the collections and the map interface hierarchy (in Java 1.8)*. There was no limit in time to do it (the project lasted during several weeks during which they had other courses and projects). An introductory course on Formal Concept Analysis presented the theory and its application to superclass refactoring. They worked in 11 groups of 3-4 persons.

They first had to analyze the existing hierarchy in Java 1.8. They could use any document they want, and they had as references the Oracle tutorial[2] and available source code[3]. At the end of this initial step, they had to produce a schema of what they found in this hierarchy (data types: interfaces, abstract classes, concrete classes). They had to distinguish technical classes that were neither a Collection nor a Map representation and to indicate which data types they will use to extract interfaces.

The second step consisted in selecting the description of data types and building it. Suggestions were given: signatures including or not names, return type, parameter type list, parameter name list, exceptions, public final static attributes, constructors, other Java interfaces implemented (like `Serializable, Cloneable, ...`) but they were free in their selection. It was also suggested to apply some inference mechanisms on datatypes, for example, in return types: When a method m returns an object of type T, this object belongs to all super types of T and this can be included in the description. At the end of this step, students had to present a set of formal contexts (the format being the input format of the targeted tool), with a text explaining their choices with regard to the final objective. For extracting data, one group used JDT tools on the source code, while all other 10 groups chose to use the Java Reflect package. The reason is not clear. They had been given an example of an incomplete program using the Java Reflect package, because the course on that package was given two years before, during their bachelor studies. The course and another practical work about JDT tools were given a few weeks before the project. Thus it was a recent notion for them. With the Reflect package, they could not have information about generic types, due to the Java policy of type erasure, but even the group which used JDT tools did not use generic type information.

The third step consisted in using RCAexplore[4] for building the AOC-posets. A little documentation and an example were given to help them using the tool. RCAexplore is a tool which implements several algorithms for Formal Concept Analysis applications, and particularly Relational Concept Analysis (RCA) which is one of the extensions of FCA to several sets of objects described by attributes and relationships to other objects. The tool input is a file containing one or several object/attribute or object/object relations. The output is one or several conceptual structures (concept lattice, AOC-poset, iceberg lattice), like these appearing in the figures of this paper, that are presented in several formats. The XML output can be navigated with a specific browser of RCAexplore that allows the analyst to explore her/his data while going from concept to concept. Using the concept browser, or using the output graphical structure required to understand the conceptual structure. We were interested in knowing in which extent software practitioners, as our students are, would be able, after a single course of one hour and half, to manipulate, analyze and extract relevant information from the conceptual structure.

Then the fourth step was dedicated to the analysis of the results. The students had to produce a document showing the built structures and explaining what they understood. They were given a few guidelines for this step: compare the information in the AOC-poset with the initial Java 1.8 interface and class hierarchy, try to name the newly discovered interfaces, build the concept lattice to see the difference between the lattice and the AOC-poset, extract relevant implication rules from the conceptual structures (AOC-poset or concept lattice).

4. VARIATION ON THE DATASET

In this section, we present the data types selected by the students to make their interface extraction, and the description they considered. This was the most surprising result, as many variations appeared on the selected dataset, and

[2]https://docs.oracle.com/javase/tutorial/collections/

[3]http://hg.openjdk.java.net/jdk8u/jdk8u/jdk/file/5910b94ea083/src/share/classes/java/util/

[4]http://dolques.free.fr/rcaexplore/

especially on the description. In this section, we use concept lattices to represent the choices of the students, thus now formal concepts describe student groups (each student is named in the group with his/her last name initial) by other information. Thus concept extents are sets of student groups. The second information is specific and will be explained hereafter.

4.1 Studied Sets of Classes and Interfaces

Fig. 5 shows which Java interfaces the students selected. It can be observed that many (but not all) groups considered Maps (see `Concept_interf_2`), some considered concurrent collections (see `Concept_interf_0` and `Concept_interf_1`). We make the same observation on the selected Java abstract classes (Fig. 6) and concrete classes (Fig. 7). In concrete classes, where we observe more variation, some students added sorting-dedicated classes (like `TimSort`), helper classes (like `ArrayPrefixHelpers`) or internal classes (like `SubList` in `AbstractList`), which is questionable.

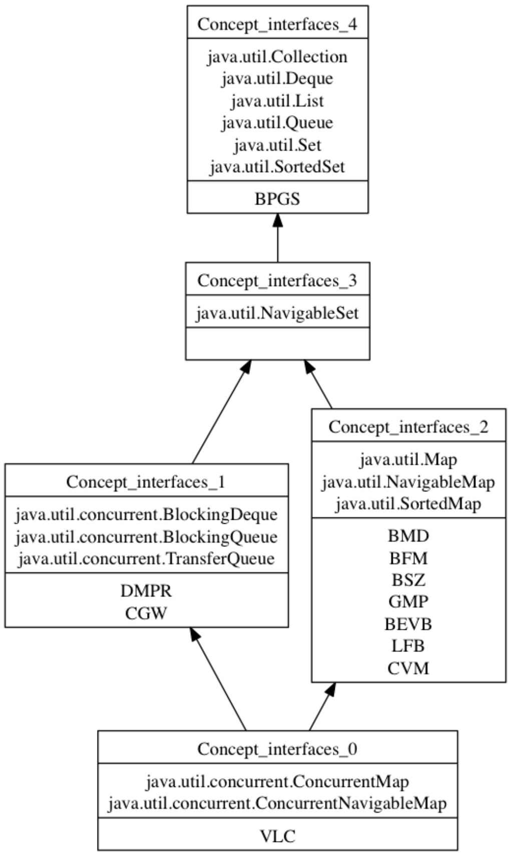


Figure 5: The studied interfaces

4.2 Selected Data Description

Here the students were even more creative, while we could have expected a similar mode of description (mainly based on public methods signatures). Fig. 4 shows the complexity of the variations. Here are some clues to read the diagram:

- "AE" represents the data types analyzed (rows of the formal contexts). E.g. `AE:concrete.classes` means that the students used concrete classes in the rows.

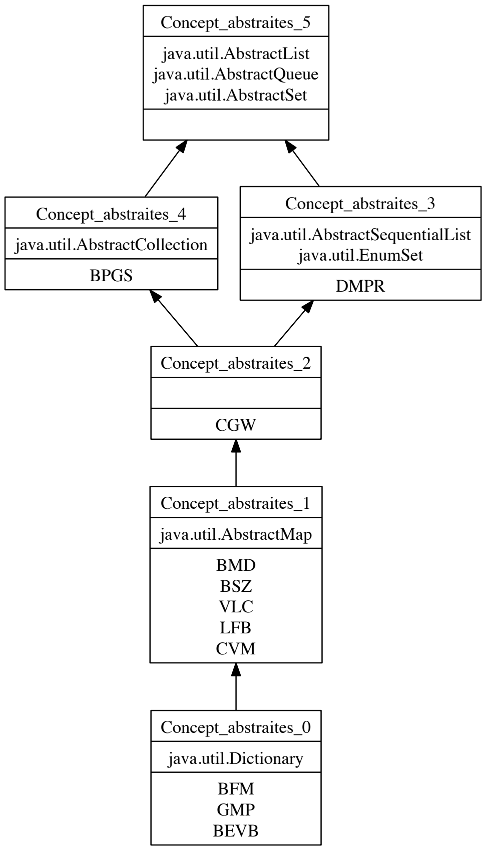


Figure 6: The studied abstract classes

- `Abs.Conc.`$\times$`.finalStaticFields.methodsNames` represents the formal context whose rows are abstract and concrete classes and whose columns are final static fields and method names.

- "itf" means "interface"; "allImplement.itf" means: all implemented interfaces appear in the columns; "implemented.itfOutOfCollections" means: only implemented interfaces that come outside the collections and maps appear in the columns (e.g. `Cloneable`, but not `List`).

- "excep" means "exception".

The figure shows that all students selected the concrete classes in at least one of their collected datasets. Many of them used (public) method signature or simply method names which already gave relevant structuring. It was surprising to see that many students considered private final static fields, that have *a priori* no relation with the interface notion. Several groups described their data types with the implemented interfaces. This has some sense, but is not clearly an element that we would introduce in an interface internal, it is rather an information to be used to connect the interface to existing ones. In addition, there is a variation about the data types that appear as rows in the formal contexts. Some groups only use concrete classes to extract the interfaces, while others also use existing Java interfaces or abstract classes.

5. VARIATION ON THE ANALYSIS

Disregarding the differences in their datasets and descriptions, the students made the same range of work during analysis. They tried to name the concepts, to recognize existing data types in the obtained AOC-posets, they remarked the high size of the lattices. Six groups did a systematic review of the built concepts, which may be painful depending of the size of the AOC-poset. Two groups tried another conceptual structure offered by the tool (Iceberg lattices). One group suggested to divide the data types by theme (as we did, by selecting only sequences) to reduce the burden of the analysis and avoiding non-relevant factorizations.

As said in the previous section, some groups introduced existing Java interfaces or abstract classes in their formal contexts. Here again, this may seem strange because the initial principle is to extract an interface from a concrete class. Nevertheless, this can be useful to observe in the conceptual structure where the existing Java interfaces and abstract classes appear.

This phenomenon can be observed with the help of Fig. 3 (the conceptual structure built from classes and interfaces can be seen on the webpage of additional documents, see below). For example, `Concept_Coll_9` in Fig. 3 does not reveal a new interface, it corresponds to the existing Java `List` interface which is rediscovered by the approach. `Concept_Coll_10` in Fig. 3, which introduces `peek`, can be interpreted, on the contrary, as a new data type, namely the type of stack-spirit collections whose top object can be consulted.

6. DISCUSSION

The students also had an individual examination based on that project (in limited time and without communication with other students). This allowed to confirm what was highlighted in their work. Except for a few students, the individual examination showed that they very well understood the concept lattices and AOC-posets. They had to understand and comment the figures that are included in this paper and this did not pose any problem to the majority of the students. This was not expected, because we thought that the conceptual structures could have been difficult to apprehend for them. They used the conceptual structures more as an analysis tool than as a turn key solution for extracting an interface hierarchy, which is an interesting deviation of the initial exercise. They did not discuss about the way the features (signatures, attributes) are grouped inside the concepts (but the question was not explicitly asked).

The diversity in analyzed data showed that, to use AOC-posets for FAEIR, the practitioner has to be guided along several directions: in the choices of the initial data types (for using rather concrete data types in a thematic subset) and in the choice of the description (1) by focusing on what will really be inside an interface, and discarding other implemented interfaces, private members, etc., (2) by suggesting several parameterizations, as including or not thrown exceptions in the signature, genericity, whole signatures or just method names, etc. This will guide our future experiences. As noted by an anonymous reviewer, a practitioner will probably not make the effort than Master students did to analyze several results and select the most appropriate one, thus a tool should help her/him to focus on what she/he is seeking for.

The reviewers also pointed out some bias in this experiment that are to be said. Students could interact during the project, even if, looking at the variety of the results, if they interacted, this seems to have had no significant impact. Besides, students organized themselves to build the groups, which may have resulted in homogeneous groups (with similar point of views and similar motivation).

Nevertheless, it was interesting to observe how the students managed this project, their autonomy, their findings and that there are different possible tracks to exploit the observation to design a tool for FAEIR application.

7. ACKNOWLEDGMENTS

The author warmly thanks the master students (Master in software engineering studies, last year, Montpellier University, finishing in july 2016) who did the experience with care and thoroughness. She also would like to thank the anonymous reviewers of IWoR 2016 for their constructive remarks that allowed to improve the paper.

8. ADDITIONAL DOCUMENTS

The input files of the students for RCAexplore, that show which data they extracted, the AOC-poset built from interfaces and classes of Java 1.8 sequences and the instructions that were given to the students (translated from french) are available here: https://www.lirmm.fr/users/utilisateurs-lirmm/marianne-huchard/publications/iwor

9. REFERENCES

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 2012.

[2] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag New York, Inc., 1999.

[3] R. Godin and H. Mili. Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices. In *Proc. of OOPSLA1993*, pages 394–410, 1993.

[4] R. Godin, H. Mili, G. W. Mineau, R. Missaoui, A. Arfi, and T. Chau. Design of Class Hierarchies Based on Concept (Galois) Lattices. *TAPOS*, 4(2):117–134, 1998.

[5] M. Huchard, H. Dicky, and H. Leblanc. Galois lattice as a framework to specify building class hierarchies algorithms. *ITA*, 34(6):521–548, 2000.

[6] M. Huchard and H. Leblanc. Computing Interfaces in Java. In *Proc. of ASE 2000*, pages 317–320, 2000.

[7] K. Lano and S. K. Rahimi. Case study: Class diagram restructuring. In *Proc. Sixth Transformation Tool Contest, TTC 2013*, pages 8–15, 2013.

[8] G. Stumme, R. Taouil, Y. Bastide, N. Pasquier, and L. Lakhal. Computing iceberg concept lattices with T. *Data Knowl. Eng.*, 42(2):189–222, 2002.

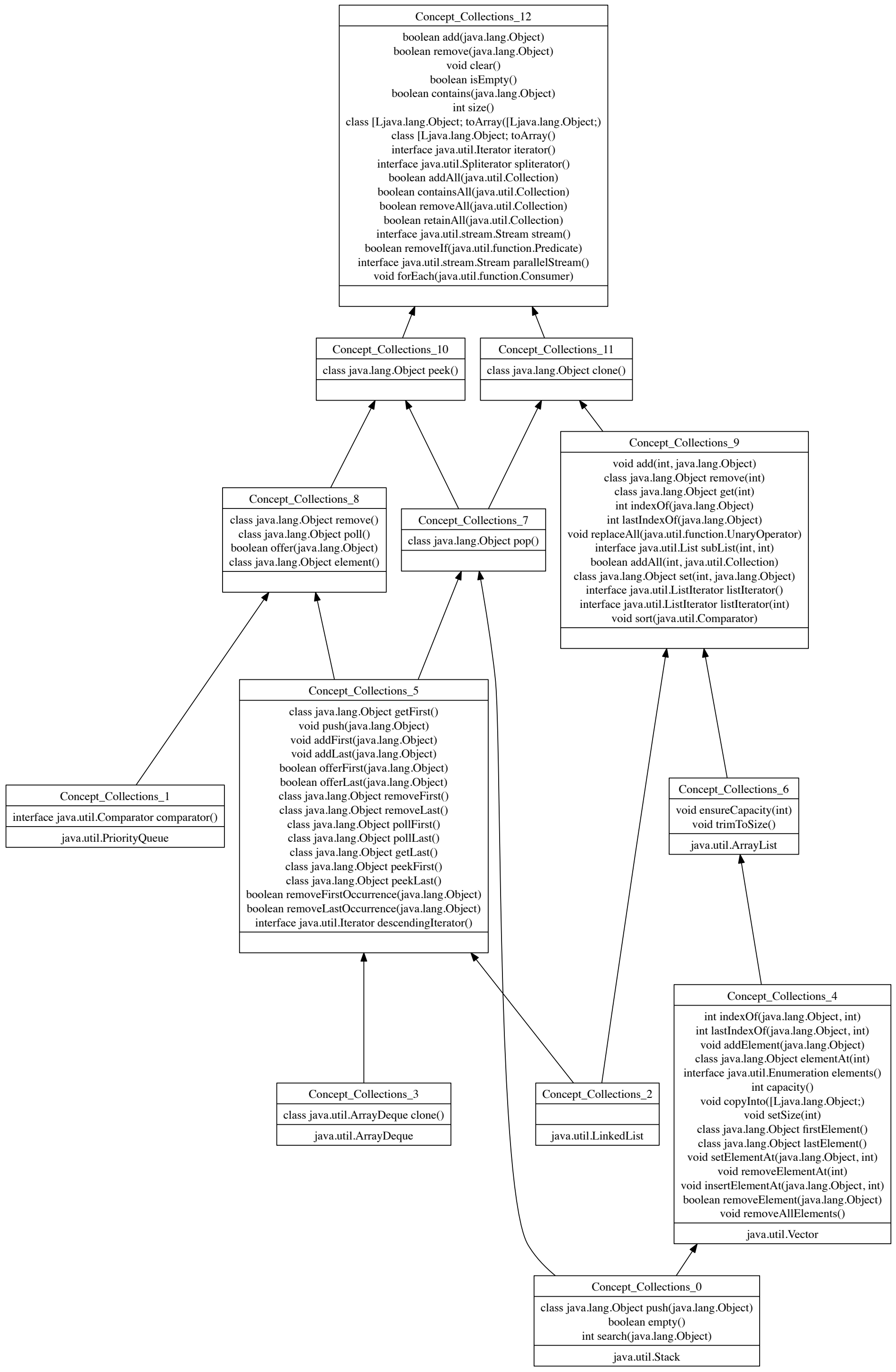

Concept_Collections_12
boolean add(java.lang.Object)
boolean remove(java.lang.Object)
void clear()
boolean isEmpty()
boolean contains(java.lang.Object)
int size()
class [Ljava.lang.Object; toArray([Ljava.lang.Object;)
class [Ljava.lang.Object; toArray()
interface java.util.Iterator iterator()
interface java.util.Spliterator spliterator()
boolean addAll(java.util.Collection)
boolean containsAll(java.util.Collection)
boolean removeAll(java.util.Collection)
boolean retainAll(java.util.Collection)
interface java.util.stream.Stream stream()
boolean removeIf(java.util.function.Predicate)
interface java.util.stream.Stream parallelStream()
void forEach(java.util.function.Consumer)
Concept_Collections_10
class java.lang.Object peek()
Concept_Collections_11
class java.lang.Object clone()
Concept_Collections_9
void add(int, java.lang.Object)
class java.lang.Object remove(int)
class java.lang.Object get(int)
int indexOf(java.lang.Object)
int lastIndexOf(java.lang.Object)
void replaceAll(java.util.function.UnaryOperator)
interface java.util.List subList(int, int)
boolean addAll(int, java.util.Collection)
class java.lang.Object set(int, java.lang.Object)
interface java.util.ListIterator listIterator()
interface java.util.ListIterator listIterator(int)
void sort(java.util.Comparator)
Concept_Collections_8
class java.lang.Object remove()
class java.lang.Object poll()
boolean offer(java.lang.Object)
class java.lang.Object element()
Concept_Collections_7
class java.lang.Object pop()
Concept_Collections_5
class java.lang.Object getFirst()
void push(java.lang.Object)
void addFirst(java.lang.Object)
void addLast(java.lang.Object)
boolean offerFirst(java.lang.Object)
boolean offerLast(java.lang.Object)
class java.lang.Object removeFirst()
class java.lang.Object removeLast()
class java.lang.Object pollFirst()
class java.lang.Object pollLast()
class java.lang.Object getLast()
class java.lang.Object peekFirst()
class java.lang.Object peekLast()
boolean removeFirstOccurrence(java.lang.Object)
boolean removeLastOccurrence(java.lang.Object)
interface java.util.Iterator descendingIterator()
Concept_Collections_1
interface java.util.Comparator comparator()
java.util.PriorityQueue
Concept_Collections_6
void ensureCapacity(int)
void trimToSize()
java.util.ArrayList
Concept_Collections_4
int indexOf(java.lang.Object, int)
int lastIndexOf(java.lang.Object, int)
void addElement(java.lang.Object)
class java.lang.Object elementAt(int)
interface java.util.Enumeration elements()
int capacity()
void copyInto([Ljava.lang.Object;)
void setSize(int)
class java.lang.Object firstElement()
class java.lang.Object lastElement()
void setElementAt(java.lang.Object, int)
void removeElementAt(int)
void insertElementAt(java.lang.Object, int)
boolean removeElement(java.lang.Object)
void removeAllElements()
java.util.Vector
Concept_Collections_3
class java.util.ArrayDeque clone()
java.util.ArrayDeque
Concept_Collections_2
java.util.LinkedList
Concept_Collections_0
class java.lang.Object push(java.lang.Object)
boolean empty()
int search(java.lang.Object)
java.util.Stack

Figure 3: Conceptual structure extracted from 6 main concrete classes of sequences of Java 1.8

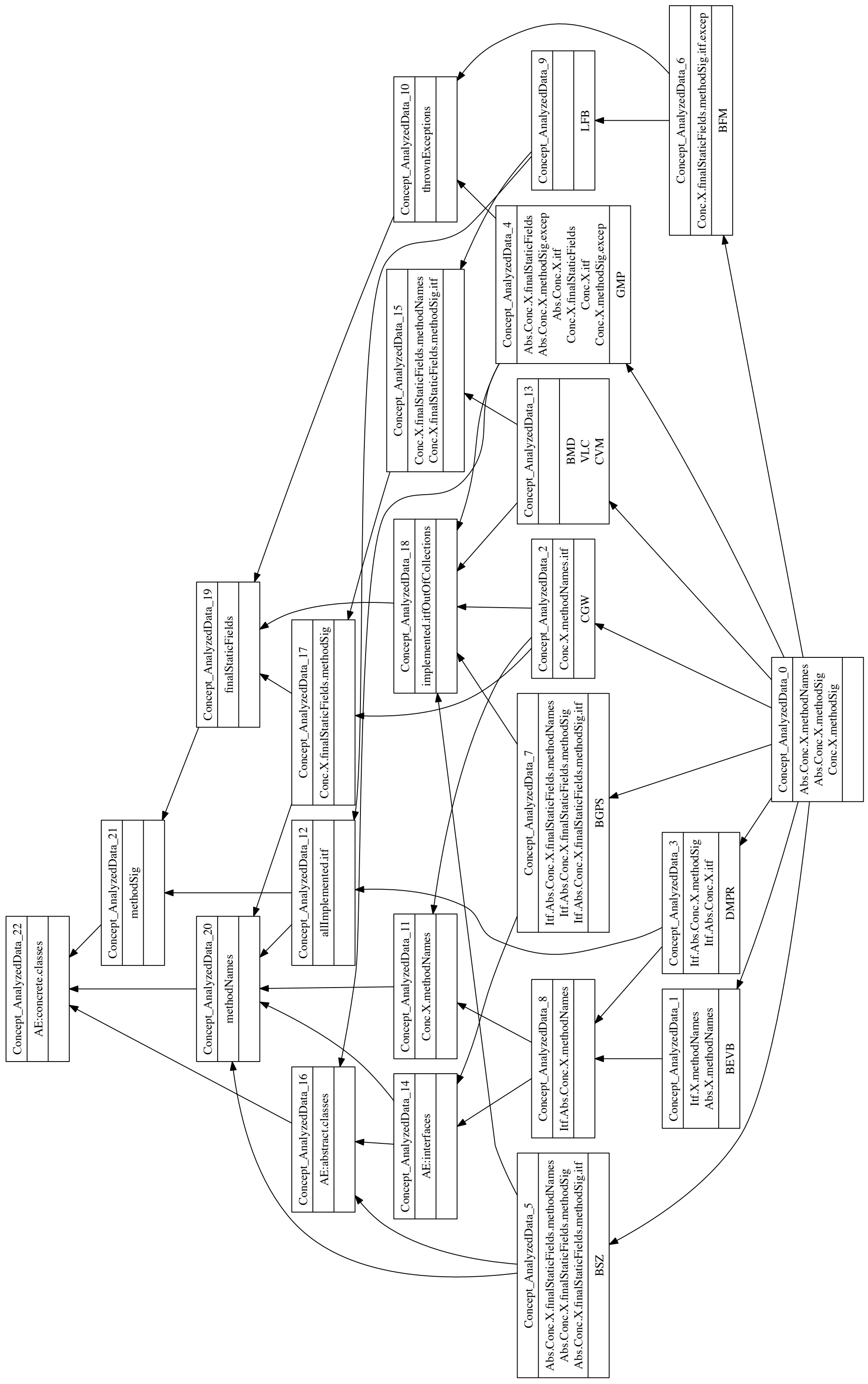


Figure 4: Variation in the description

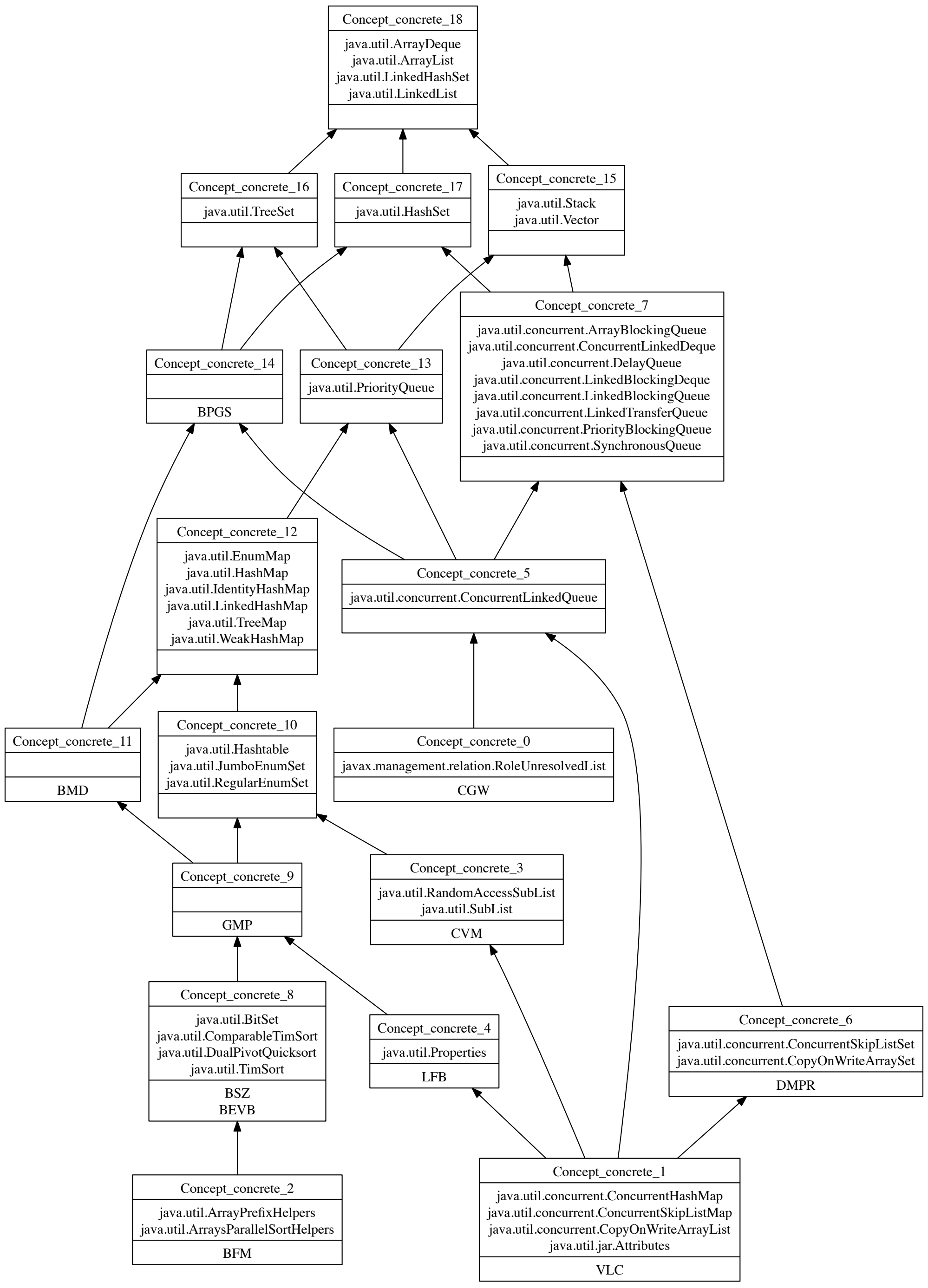


Figure 7: The studied concrete classes