IRIS AperTO

UNIVERSITÀ DEGLI STUDI DI TORINO

AperTO - Archivio Istituzionale Open Access dell'Università di Torino

A Trait Based Re-engineering Technique for Java Hierarchies

This is the author's manuscript

Original Citation:

Availability:

This version is available http://hdl.handle.net/2318/34229 since 2016-03-11T17:47:16Z

Publisher:

ACM

Published version:

DOI:10.1145/1411732.1411753

Terms of use:

Open Access

Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)

23 April 2024

A Trait Based Re-engineering Technique for Java Hierarchies*

Lorenzo Bettini[1] Viviana Bono[1] Marco Naddeo[1]
[1]Dipartimento di Informatica, Università di Torino

ABSTRACT

Traits are pure behavior components introduced in the Smalltalk community in order to integrate the traditional class inheritance with a composition mechanism: a class is composed by traits and inherits from superclasses. This offers the advantage of promoting code reuse. In this paper, we tackle the problem of re-engineering a Java hierarchy into traits, by adapting to a Java setting a methodology developed by Lienhard, Ducasse, and Arévalo for a Smalltalk setting, based on Formal Concept Analysis. We illustrate the approach by applying it to the Java input stream library. We also obtain two by-products: (i) we identify clearly some workarounds that programmers must exploit in order to overcome some of the limitations of Java single inheritance; (ii) we single out some features a Java with traits might include, as none of the proposals in the literature in this sense has taken the lead yet.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Design, Theory

Keywords

Code Reuse, Re-engineering, Java, Trait, Formal Concept Analysis

1. INTRODUCTION

It is well-known that single inheritance in object-oriented languages has some intrinsic limitations. Often the restriction of having only one direct superclass forces programmers to some design choices that are not optimal from the point of view of the domain conceptual representation, of the understandability of the code, and especially from the point of view of the code reuse, which is, in principle, one of the main goals of the object-oriented paradigm. In the literature, there are some proposals to overcome such limitations. Notably, multiple inheritance is supported in different forms by languages such as C++ [32] and Eiffel [22]; another interesting alternative construct is the mixin based inheritance [8, 9, 10, 2, 16]. Nevertheless, none of them is completely satisfactory, in fact none of them has ever taken the lead completely as a substitute of single inheritance (for a full account on the subject we refer to [12]).

A new construct called *trait* was introduced in the Smalltalk community, in order to complement single inheritance and to solve some of the problems introduced by it. Traits [29, 12] are pure units of reuse, constituted of methods only, that can be used as building blocks to compose classes. The most important feature of traits is that the composed class has a complete control on the composition itself, which implies, in particular, that traits and their composition are orthogonal to any inheritance hierarchy; moreover, it is the programmer that must solve explicitly any conflict that may arise when composing the traits. Traits have been incorporated in Squeak [18, 3], then in other languages such as Scala [25] and Fortress [1], in a typed context [14], and in the Java setting [31, 21, 7].

When traits are introduced in a language, either to complement traditional single inheritance or as the only construct to build classes, it is of interest to tackle two (related) aspects: (*i*) providing techniques and tools as aids for programming fruitfully with the new construct from scratch (see, for instance, [26], in which a tool for supporting Java traits in Eclipse is presented); (*ii*) supplying special *code refactoring* procedures and tools, to transform legacy code under the form of single-inheritance based hierarchies into trait based hierarchies, in order to improve the reuse possibilities of the legacy code itself.

In this paper, we present a semi-automatic technique for refactoring a traditional Java hierarchy into groups of traits We have chosen not to select a specific "Java with traits" target language among the proposals in the literature [31, 21, 7], for three reasons: (*i*) there is still not a proposal which prevails on the others, neither in the literature, nor in the "outside world"; (*ii*) our goal is to suggest simply a possible set of traits (as groups of methods) that can be used to compose classes that can be integrated within other, orthogonal, programming constructs; (*iii*) we also want to discuss on which would be a good candidate to become the "Java with traits", that is, which would be the constructs useful to complement traits. A by-product of our work is to single out some of the workarounds programmers must apply in order to overcome some of the drawbacks of single inheritance. Nevertheless, traits and trait composition alone are not enough. In particular, in order to preserve the polymorphism present in a single-inheritance based hierarchy, we need at least to have the interface programming construct (in the sense of Java). For a discussion on other possible combinations of traits with other Java constructs, see Section 5.

*This work has been partially supported by the MIUR project EOS DUE.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPPJ 2008, September 9–11, 2008, Modena, Italy.
Copyright 2008 ACM 978-1-60558-223-8/08/0009 ...$5.00.

Our refactoring technique is based on the approach proposed by Lienhard, Ducasse, and Arévalo [19] for Smalltalk that exploits Formal Concept Analysis [17], a solid mathematical theory for gathering significative groupings of *elements* which have some *properties* in common. The main novelty is that we apply this technique to Java, which is a typed language (while Smalltalk is untyped). We worked on the Java library defining input streams (shown in Figure 1) as a case study. We also implemented a substantial part of a tool that performs the trait-based refactoring as an Eclipse plug-in.

The paper is based on the content of the master thesis of Naddeo [23] and it is organized in the following way: Section 2 illustrates, via our case study, some of the design choices forced by single inheritance, Section 3 gives a short introduction on traits, Section 4 presents our refactoring approach from single-inheritance based hierarchies to trait based ones in detail, and Section 5 draws some conclusions on the work done.

2. SOME DRAWBACKS OF SINGLE INHERITANCE AND SOME SOLUTIONS

Despite the fact that the single inheritance mechanism has been introduced as one of the key factors in object-oriented programming, especially for improving code reuse, this mechanism has its limitations.

One of the most frequent mandatory design choices when using single inheritance is *code duplication*, often in alternative or combined with the strategy of implementing some *methods higher in the hierarchy* (where they might not be conceptually needed, but this way it is possible to make them inherited by all subclasses without duplicating them), as in some situations better design choices are not possible. Let us consider as an example the following situation: classes `C1` and `C2` both extend the class `S`, inheriting all its methods; class `C1` extends `S` by introducing a new method `method1()`, and `C2` extends it by introducing a method `method2()`. We then assume to have a class `C3` that needs from the conceptual point of view to inherit both `method1()` of `C1` and `method2()` of `C2`. Since single inheritance permits inheriting from one class only, class `C3` must extend only one of those classes (for example, `C2`), therefore the method `method1()` of `C1` must be duplicated (a similar hierarchy is depicted in Figure 3, where method `next` is duplicated).

An alternative solution to code duplication is to move a method higher up in the hierarchy until it is available for all the (sub)classes that need it, but sometimes "higher" is "too high". In the example, we can move `method1()` from class `C1` into its superclass `S`, in such a way it is inherited by `C1`, `C2`, and also `C3`, avoiding code duplication. However, this has a price: the superclass `S` is "polluted" with a method that conceptually does not belong to it and the same holds for `C2`, that inherits a method that does not need.

A variant of the strategy of pushing higher up a method in a hierarchy is to implement all common methods to a set of classes into a common superclass, then to *cancel* them in the (sub)classes that do not need them by redefining them: either (*i*) with an empty body; or (*ii*) with a single instruction to throw an appropriate runtime exception (later in this section we will give some concrete examples of canceled methods present in our case study). However, such a strategy is not always applicable: in our example, we cannot cancel `method1()` in `C2`, otherwise `C3` would inherit the canceled version, making useless the moving of `method1()` from `C1` to `S`.

We could also think of an *abstract* method as a special form of canceled method, almost equivalent to a method with an empty body, however, there is a subtle, but important, difference, as the former does not have a body and cannot be invoked, while for the latter (being a concrete method) is the opposite. We will use the word "method" both for concrete and abstract methods, whenever the difference is not important.

Now we are ready to show some concrete examples of the workarounds adopted by the programmers we mentioned earlier, duplicated methods and canceled methods. We consider the Java library defining input streams (shown in Figure 1).

It is important to notice that in order to speak of duplicated methods we need two methods with identical signature and identical *behavior* implemented (not inherited) by two different classes of the same hierarchy. As a duplicated method, we consider the example: `public int read(byte[] b, int off, int len)`.

This method is implemented firstly in the hierarchy root, `InputStream`. Its direct subclasses (see Figure 1) redefine the method with a new, specific, implementation. But class `LineNumberInputStream`, needs the original root version, therefore the only possibility is to duplicate the implementation of `InputStream`.

Note that it is not always possible to detect duplicated methods by comparing syntactically the source code. The best methodology is to compare the bytecode, as suggested in [19].

As already pointed out, a canceled method can be either a method that "does nothing" or a method that does only one thing, i.e., it throws an exception. In the first case, the body can be literally empty, if the return type is `void`, or it can contain only a single return statement returning an appropriate constant value according to the return type (for instance, `null`, in the case of an object type). In the second case, the exception must be unconditioned, i.e., it should not appear in the branch of a conditional statement. We then say that a method is *directly canceled* if it has an empty body or it throws an unconditioned exception. We say also that a method is *indirectly canceled* if it contains a method call on `super` or on `this` that reaches eventually, maybe via a chain of calls, a directly canceled method. We will use the term *canceled* whenever it will not be necessary to distinguish between directly canceled and indirectly canceled. It is important to observe that our definition of *canceled method* is a pure syntactical one, therefore any automated analysis that detects canceled methods must be semi-automatic, in the sense that its results must be approved by the user. This is what our tool does.

As examples of directly canceled methods, we consider the following ones: `public void mark(int readlimit)`, `public void reset()`, `public boolean markSupported()`.

The method `mark` marks the current position in the input stream, in such a way that the next call to `reset` will reposition the stream to that position, in order to make available the same bytes. The argument `readlimit` of `mark` represents the maximum number of bytes that it is possible to read before the marked position becomes invalid. Therefore, `mark` and `reset` are two methods to be used together. The problem is that not all the types of input stream of the hierarchy of Figure 1 support such operations; in other words, the fact that `mark` and `reset` are supported is an invariant property of any particular type of input stream. As a consequence, the goal of `markSupported` is to flag whether a stream supports the method pair `mark` and `reset`, by returning either `true` or `false` accordingly. To understand better, we examine the contracts of `mark` and `reset` in some detail:

- `mark`: if the method `markSupported` returns `true`, the stream takes into account all the bytes read after a call to `mark` and it can make them available again any time `reset` is called. However, the stream does not take into account the readings when more than `readLimit` bytes are read from the stream before a `reset`

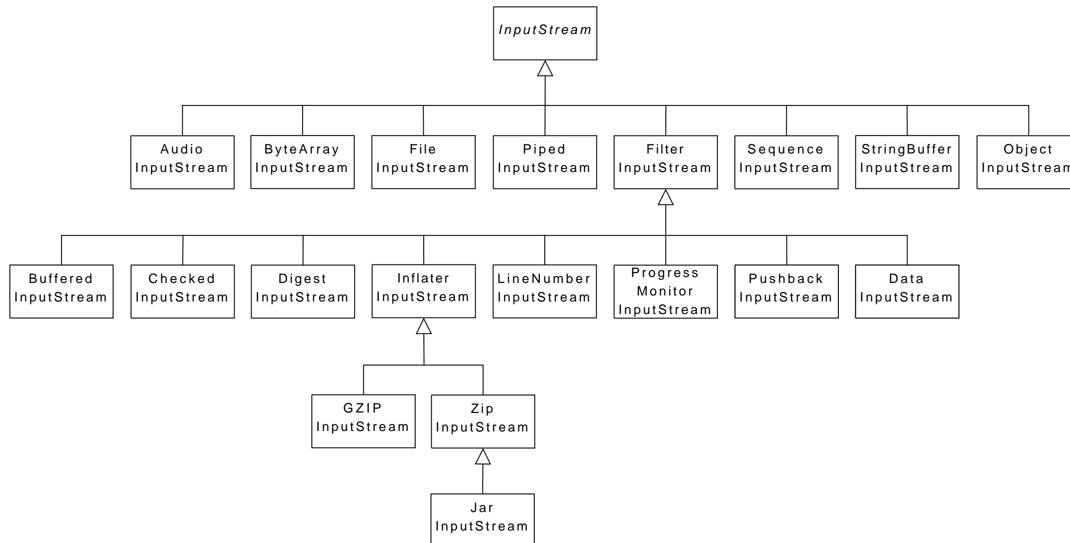

Figure 1: The Java input stream hierarchy.

invocation.

- `reset`: its contract is:
  - if `markSupported` returns `true`, then:
    - if `mark` was never called since the stream was created, or the number of bytes read by the input stream is bigger than the last `readLimit`, an `IOException` may be thrown;
    - if such an exception is not thrown, then the input stream is reset to a state in which all the bytes read starting from the most recent call to `mark` (or starting from the beginning of the stream, if `mark` has not been called) are made available again by calling `read`, followed by all the bytes available before calling `reset`.
  - if `markSupported` returns `false`, then:
    - a call to `reset` may throw an `IOException`;
    - if such an exception is not thrown, then the stream is reset to a fixed state that depends on the type of input stream and the way it was created. The returned bytes by `read` depend on the type of input stream.

Summarizing, if `markSupported` returns `true`, then it is possible to use `mark` and `reset` in pair, otherwise a call to `mark` will not have any effect, while a call to `reset` either will throw an exception or it will reset the stream to a certain state depending on the type of stream (for instance, `reset` of class `StringBufferInputStream` repositions the stream at the start of the given string).

These methods are redefined as canceled in the classes that do not support them: `InputStream`, `InflaterInputStream`, and `PushbackInputStream`. In particular the last two must cancel them again, since their superclass `FilterInputStream` overrides the original canceled versions of `InputStream`. We note that while `mark` is canceled with an empty body, `reset` is canceled with an exception. This difference is a consequence of the fact that `reset` may have an effect even in the case `markSupported` returns `false` (that is, it may reset the stream to a given state), therefore there is the necessity of throwing an exception when `reset` is *really* not supported; if it were canceled by implementing it with an empty body, we would not have a way to determine if a call to `reset` has an effect or not in the case when `markSupported` returns `false`.

We finish this discussion on the canceled methods of the input stream hierarchy by noting that:

- the canceled implementations of `mark` and `reset` defined in `InputStream` are inherited by the classes `FileInputStream`, `ObjectInputStream`, `PipedInputStream`, and `SequenceInputStream`;
- similarly, the implementations of the two methods defined in the class `InflaterInputStream` are inherited by the classes `GZIPInputStream`, `ZipInputStream`, and `JarInputStream`;
- class `StringBufferInputStream` inherits the canceled implementation of `mark` from `InputStream`, but it redefines `reset` in such a way the stream is repositioned at the beginning.

3. AN INTRODUCTION TO TRAITS

The trait construct was proposed in the Smalltalk community [29, 12], and implemented for the first time in Squeak [18, 3], a Smalltalk dialect; other proposals and variants appeared afterward in the literature, such as the ones in [15, 31, 27, 28, 6, 21], but we will base our short introduction on the seminal work.

A trait is essentially a set of methods, completely independent from any class hierarchy. They are building blocks to compose classes or other, more complex, traits. They can be seen as *units of reuse*, as the common behavior, that is, the common methods, of a bunch of classes can be factored into a trait.

A trait can consist of *provided methods*, that implement their behavior, and of *required methods*, that parametrize the behavior itself. The original traits do not specify any state, therefore one of the goals of the required methods is to tie the defined methods to the state of a class that will use the traits (such a class will have to implement the required methods, often called *glue methods*).

Trait composition is the main operation to build classes and composite traits and the programmer has a complete control over possible conflicts that can arise when composing traits. In the original proposal, trait composition is seen as complementary to single inheritance, therefore classes are still organized in a hierarchy, even though they can exploit traits to specify their own behavior, with respect to their superclass. This approach has the following consequences: (*i*) roles are separated: classes are object generators and

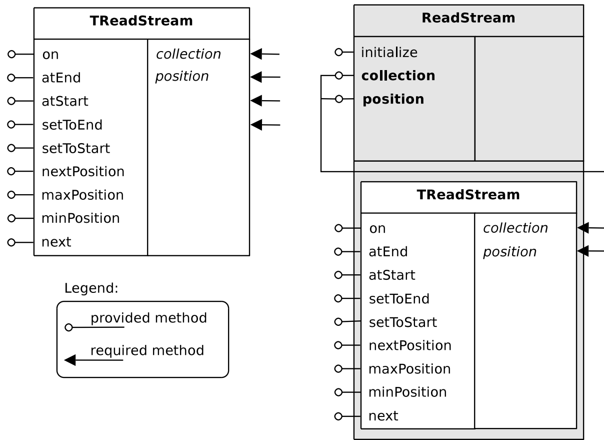

Figure 2: The trait `TReadStream`, with its *provided* and *required methods*, and the class `ReadStream`.

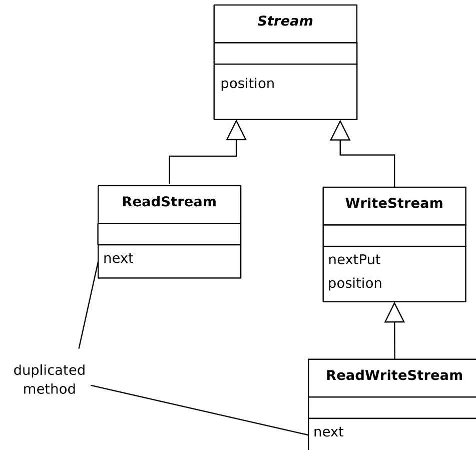

Figure 3: Code duplication in the Smalltalk stream hierarchy.

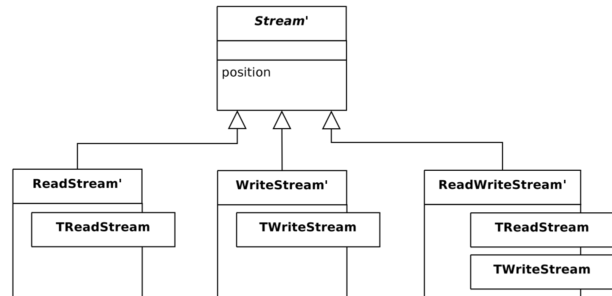

Figure 4: The Smalltalk stream hierarchy re-factorized.

traits are units of reuse; (*ii*) since their semantics is simple, it is easy to give a classical single-inheritance based semantics to a trait based hierarchy via a so called *flattening property* [24, 12], that states that a hierarchy with traits is always equivalent to a hierarchy without traits, via a sound translation function; (*iii*) the problems that arise with multiple inheritance do not arise with traits as they are independent from any form of inheritance.

Since traits do not have a state (this is still true in most of the proposals in the literature), the only kind of conflict that can arise during composition is among methods, but this can be always avoided by using some operators, such as *overriding* (that gives a method a new implementation, hiding the old one) and *exclusion* (that hides a method from the external traits). Another useful operator is *aliasing*, that gives an additional name to a provided method.

A class can be composed by one or more traits, can inherit (part of) its state from a superclass and implements the glue methods. A class is *complete* if all the required methods of the composing traits are part of the glue methods, or are inherited, or are supplied by other traits. A class formed via traits must specify the composing traits via a *composition clause*. Note that also composite traits must define a composition clause and can use all the above operators.

An example. We borrow the following example from the Squeak community. We want to implement a library of streams that can be readable, writable, readable/writable, or synchronizable. In the sequel of the example, we will use for traits names beginning with `T`, *italics* to emphasize the required methods, and **boldface** to emphasize the glue methods.

We build the library starting from some basic traits, `TReadStream`, `TWriteStream`, `TSynchronize`. To present them, we use an UML-like syntax, see Figure 2. The figure also shows how the class `ReadStream` is created by using the trait `TReadStream`, parametrized via its required methods: `collection` and `position`[1]. Class `ReadStream` uses trait `TReadStream`, implements the required methods of `TReadStream` (in addition to method `initialize`), it is a subclass of `Object`, and it has two instance variables (`position` and `collection`).

Since traits `TReadStream` and `TWriteStream` have methods in common, we can factor them into a new trait, `TPositionableStream`, that offers the operations to manipulate a position on a collection. Traits `TReadStream` and `TWriteStream` can be then defined as composite traits that use `TPositionableStream`.

[1]For simplicity, we only write method names, without their complete signature.

Why traits are useful. Traits can improve code reuse, as they can be exploited to factor duplicated methods into a trait: all classes needing those methods declare the trait in their composition clause. Moreover, the form of reuse based on traits is complementary to any form of reuse introduced by any kind of inheritance. If we give to the trait mechanism the burden of dealing with the low-level reuse, inheritance can be fruitfully exploited to model the conceptual relations among classes. This is noticeable, for example, from Figures 3 and 4, that show a part of the Smalltalk stream hierarchy, respectively in a single-inheritance based version and in a trait based one. In the version without traits, it is possible to see that the conceptual relation between classes is not well modeled: class `ReadWriteStream` is connected conceptually to class `WriteStream`, but not to `ReadStream`, moreover, since `ReadWriteStream` should be subclass of both `WriteStream` and `ReadStream` (forbidden in single inheritance), such a class must duplicate the behavior (method `next`) instead of inheriting it from `ReadStream`. All these drawbacks are avoided in the trait based hierarchy, that maximizes code reuse and it is conceptually sound at the same time. We refer to [29, 12] for a comparison between traits and other constructs alternative to single inheritance.

4. THE FCA TECHNIQUE APPLIED TO JAVA

In this section, we want to illustrate how to apply the Lienhard-Ducasse-Arévalo approach [19], born in the Smalltalk setting, to a Java setting. In particular, we apply the Formal Concept Analysis (FCA) to refactor a Java hierarchy into a trait based one. Our case study is the input stream hierarchy (see Figure 1). For "FCA in a nutshell" see the appendix of [19], and for a full account on FCA see [17].

Our plan is to offer a (semi-)automatic refactoring technique possibly to complement a completely manual technique (that was applied, for example, in [5] for some significant portions of the Squeak library, or in [13] for the *kernel* of Squeak itself).

The main idea of the approach is to single out from a class-based hierarchy those groupings of methods that are potentially good candidates to be factored into traits, which will be re-composed afterwards into an equivalent hierarchy (for some considerations on the semantical equivalence see Section 5). It is important to notice that this approach is purely a "curative" one, in the sense that its goal is to find the traits that are necessary to improve the reuse potential of a hierarchy by eliminating the problems we described in Section 2, that is, code duplication and methods implemented too high in the hierarchy. This is already highlighted in [19], where the original Smalltalk version of the technique is applied to the Squeak stream and collections libraries: they were able to find a trait-based hierarchy that solved the major problems of reusability, but such a hierarchy has less granularity from the point of view of the number of traits than the manual solution presented in [4]. However, this is not to be considered a flaw of the approach, as it is based on an analysis of the structural relations between classes and methods, not on the semantics of the methods themselves, and it can be an effective complement to manual techniques.

4.1 Identifying traits using FCA

By using FCA, we can identify some significant (*maximal*) groupings of *elements* (that is, classes) that enjoy certain *properties* (that is, methods that are "meaningful" for certain classes, see below for a thorough explanation of what we intend). The groupings are called *concepts* and they can be sorted into a *concept lattice* following a partial order based on the relation *subconcept-superconcept*. This relation is fundamental in our refactoring as it gives indications on how to find traits as groupings of methods, how they can be composed, and which classes must exploit them in order to maximize reuse in a way completely orthogonal to the single inheritance chain.

Our approach follows closely the original one [19] and is structured in two phases. The first phase restructures the hierarchy by identifying the principal traits. The second phases decomposes and refines further the hierarchy built in the first phase. In the first phase, we identify the set of methods which are "meaningful" for each class. This is the input to the FCA, that finds the groupings of methods shared by a certain set of classes; therefore, starting from these groupings, we can identify the traits and refactor the original hierarchy. In the second phase, FCA is applied again to each *single* class and each *single* trait to identify more refined significant groupings of methods on the basis of an analysis on method invocations.

The refactoring approach is organized in six steps shown in Figure 5. Steps 1 to 4 represent the first phase of the process, Step 5 and 6 are the second one. Step 1 generates the input for the FCA by analyzing the classes of the hierarchy in question. Step 2 uses FCA to produce the *concept lattice*. Step 3 elaborates the lattice to eliminate the superfluous information. Step 4 deduces the classes to be implemented and the traits that compose them. Moreover, it infers the interfaces that must be introduced in order to recover the polymorphism of the original hierarchy, being in a typed setting (in fact, in the original untyped setting there is no need to have any form of inheritance along with trait composition). Step 5 applies FCA again to each single class and trait found in the previous steps, and produces the corresponding concept lattices. Step 6 decomposes the traits found in Step 5 in sub-traits. In this work, we concentrated on the first four steps, described below in detail.

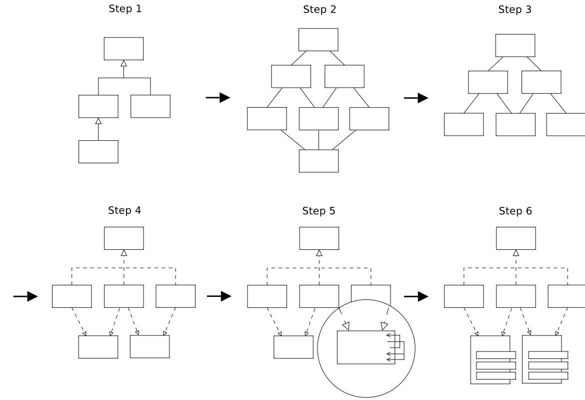

Figure 5: A glance to the approach (picture borrowed from the original work.

Step 1: finding out meaningful methods. This step is based on the detection of those problematic methods (duplicated methods and methods too high in the hierarchy) we illustrated in Section 2. The idea is (*i*) to detect duplicated methods in order to factor them into traits that will be part in the composition of more than one class, and (*ii*) to find canceled methods (canceled methods are defined in Section 2) in order to group, for each class, only the methods that are really "meaningful" for them.

We say that a method is *not meaningful* for a class if (*i*) the method is abstract, or (*ii*) the method is canceled (following the terminology of Section 2), while a method is *meaningful* in all the other cases.[2]

For the purpose of detecting not meaningful methods, we do not need to analyze the method bodies in details, it is only necessary to track the method calls on `this` and on `super`. To do this, we apply a *static* analysis of the code:

- We detect abstract methods and directly canceled methods.
- We detect methods that are indirectly canceled, that is, with their body containing at least a method call on `this` or on `super` that leads eventually to a directly canceled method (we do this by following the hierarchy upwards, starting, respectively, from the class in exam, or its superclass).
- We are not interested in calls on objects other than `this` because what we want to detect are the canceled methods of the `this` class, which is the class in exam. Moreover, we are not interested in the possible dynamic bindings of `this` because what we are detecting are the canceled methods of the class in exam, not the

[2]In the original paper [19] they use the term *understood* instead of *meaningful*. We changed terminology in order not to get confused with the more common use of "understood" in phrases such as "message-not-understood", already part of the object-oriented jargon.

ones of its subclasses (these will be detected when each subclass will be the actual class in exam).

We can now model our *context*, that is, the structure needed by the FCA to produce its output: a *context* is a triple $\mathbb{C} = (\mathscr{E}, \mathscr{P}, \mathscr{R})$, in which $\mathscr{E}$ is the set of elements, $\mathscr{P}$ is the set of properties, and $\mathscr{R}$ is a binary relation between $\mathscr{E}$ and $\mathscr{P}$. In our case:

- The set of elements $\mathscr{E}$ is the set of (concrete) classes belonging to the hierarchy.

  Note that a Java hierarchy may contain also interfaces and abstract classes. We do not include interfaces in our analysis, as they obviously do not contain any meaningful method to be inserted in a trait. As for abstract classes, we decided to adopt the following approach, along the lines of having traits and interfaces only in the refactored hierarchy: we collect the concrete methods of the abstract class, put them in all the subclasses of the abstract class that do not redefine them (these methods will be detected as duplicated by our re-engineering technique and dealt with accordingly), and substitute the abstract class with a correspondent interface. In our case study, the only abstract class is the root `InputStream` (Figure 1).
- The set of properties $\mathscr{P}$ is the set of methods meaningful for at least a class belonging to $\mathscr{E}$.

  Note that, in principle, the methods inherited from `Object` (and any other superclass of the root of the hierarchy in exam) should be included in the analysis as they may be meaningful for the hierarchy. However, our objective is a local refactoring, therefore we limit the analysis to the hierarchy itself (without considering any superclass of its root). We also do not consider in our analysis private methods, as we assume they are implemented only for the class defining them, therefore they will be part of its implementation only also in the refactored hierarchy. As for duplicated methods, we already mentioned the fact that their detection should be based on a bytecode analysis, to overcome non influent differences, such as variables names, that exist instead in the source code. Duplicated methods will be considered as one element in the property set. Obviously methods with the same signature but different implementation must be considered as distinct elements of the property set.
- The relation $\mathscr{R}$ is defined as follows: a class $c \in \mathscr{E}$ and a method $m \in \mathscr{P}$ satisfies $\mathscr{R}$ if and only if m is meaningful for c.

If we apply Step 1 to our case study, we obtain a context that can be represented as an *incidence table*. A portion of the incidence table is in Figure 6. We observe that:

- The set $\mathscr{E}$ contains the classes belonging to the Java input stream hierarchy (Figure 1) except for `InputStream`, that is abstract (see above for a description of our treatment of abstract classes).
- To determine the set $\mathscr{P}$, we collected all the methods locally implemented or inherited by each class in $\mathscr{E}$. We then detected all the canceled methods and mark them as removed from $\mathscr{P}$ (see the "C"'s in Figure 6).
- Different implementations of the same signature are numbered, as they are distinct properties.
- The duplicated methods were detected and considered as a single property. Table 1 summarizes duplicated methods.
- Any "X" in the incidence table corresponds to a method which is meaningful to a class. Any "C" corresponds to a class that implements or inherits a canceled method (not to be considered in Step 2).

Method	Classes where the implementation is duplicated
`available()`	`ByteArrayInputStream` and `StringBufferInputStream`
`available()`	`InflaterInputStream` and `ZipInputStream`
`close()`	`AudioInputStream` and `FilterInputStream`
`read(byte[])`	`AudioInputStream` and `FilterInputStream`
`skip(long)`	`ByteArrayInputStream` and `StringBufferInputStream`
`skip(long)`	`InflaterInputStream` and `ZipInputStream`

Table 1: The duplicated methods in the Java input stream hierarchy.

- If there is more than an "X" for the same method, corresponding to different classes, either that method is inherited from a common superclass, or that method is duplicated (the difference is taken into account in Step 3).
- method `markSupported` (already treated at length in Section 2) deserves a discussion on its own: since it was introduced in the original hierarchy only to tell whether the pair of methods `mark` and `reset` were supported in a certain class of the hierarchy (i.e., if they were canceled or not), such a method was not included in the set of properties and it does not appear in the context.

Note that the current step cannot be automatized completely, because the technique (and any tool based on that) can detect canceled methods and their related auxiliary methods only via some syntactic criteria, but the last word on which methods are really canceled or not must be left to the user.

Step 2: generating the concept lattice. Once the context is calculated, we can apply an algorithm to generate the concept lattice. Our implementation is based on an improved version of the FCA [17], called Fast Concept Analysis [20]. Each of the individuated concepts will have a set of classes as the *extent* (set of elements) and a set of methods as the *intent* (set of properties). Since a concept represents a maximal set of elements with common properties, in our case, the concepts are maximal groupings of classes that have some methods in common. In particular, all methods belonging to a concept are: (*i*) implemented locally or inherited by all the classes in the extent; (*ii*) meaningful for all the classes in the extent.

For our case study, we obtain, starting from the context, the *concept lattice* of Figure 7. We observe that there are 19 concepts with one class, while 13 different concepts present from 2 to 16 classes in their extent. Top and bottom concepts contain, respectively, the complete set of classes and the empty set of classes as extent. In the lower part of each concept there is the list of the meaningful methods for all the classes listed in the upper part: for example, the top concept says that there is no method that is shared by all classes, while the bottom concept says that it does not exist a class that needs all the methods.

This way we know:

- The exact set of methods that a class needs either to implement, or to obtain from a trait by trait composition. In fact, the concepts with only one class in the extent, show, via their intent, the methods that such a class needs.
- The maximal sets of methods shared by different classes. In fact, the concepts that have as extent two or more classes say which are the shared methods, that is, the ones in the intent.

Step 3: elaborating the concept lattice. For each concept, we eliminate from it any method that is at least in one of its superconcepts. We obtain a structure called *gamma lattice*, whose

	getFD()	getFormat()	getFrameLength()	getLineNumber()	getManifest()	getMessageDigest()	getNextEntry()[1]	getNextEntry()[2]	getNextJarEntry()	getProgressMonitor()	~~mark(int)~~[1]	mark(int)[2]	mark(int)[3]	mark(int)[4]	mark(int)[5]	~~mark(int)~~[6]	mark(int)[7]	~~mark(int)~~[8]	nextStream()	on(boolean)	read()[1]	read()[2]	read()[3]	read()[4]	read()[5]	read()[6]	read()[7]	read()[8]	read()[9]	read()[10]	read()[11]	read()[12]	read()[13]	read()[14]
AudioInputStream		X	X									X									X													
BufferedInputStream															X														X					
ByteArrayInputStream													X									X												
CheckedInputStream														X																X				
DataInputStream														X										X										
DigestInputStream						X								X						X											X			
FileInputStream	X										C												X											
FilterInputStream														X										X										
GZIPInputStream																C																X		
InflaterInputStream																C																X		
JarInputStream					X			X	X							C																X		
LineNumberInputStream				X													X																X	
ObjectInputStream											C														X									
PipedInputStream											C															X								
ProgressMonitorInputStream										X				X																				X
PushbackInputStream																		C																
SequenceInputStream											C								X								X							
StringBufferInputStream											C																	X						
ZipInputStream							X									C																X		

Figure 6: A part of the context.

concepts' intents contain only additional methods with respect to the ones of the superconcepts. We also eliminate the bottom concept, that does not add any relevant information to our analysis (the gamma lattice for our case study is not shown here for lack of space, but it can be derived easily).

After this step of simplification, all concepts that do not have any subconcept have necessarily one class only in their respective extent (otherwise it would mean that there is a method not belonging to any class, or that there are two or more classes that define exactly the same methods).

A simple way to interpret the gamma lattice is that of considering it a multiple-inheritance based hierarchy, in which each concept represents a class that inherits from its superconcepts seen as superclasses, that is, the behavior of a class is completely defined by the methods in its concept plus the methods in its superconcepts in the gamma lattice. However, we will use the gamma lattice not to build a multiple-inheritance based hierarchy, but a trait based one.

Step 4: refactoring with traits. We analyze the gamma lattice to identify traits and classes and restructure the hierarchy with the aid of the trait composition, on the base of the following observations:

- All concepts with only one class in their extent supply, by their respective intent, the set of the methods needed only by that class: for any of such concepts, it is possible to define a class implementing those methods.
- The concepts with two or more classes in their extent constitute trait candidates; in fact, each of such concepts represent a maximal grouping of methods (in the intent) shared by some classes (in the extent), methods that can be factorized in traits composing the classes themselves.
- Each concept without subconcepts represents a class. The set of its superconcepts in the gamma lattice, direct and indirect, indicates which methods the class needs from which traits, in order to rebuilt its original behavior.
- The subconcept-superconcept gives us information on how to compose traits to obtain composite traits.
- If in the gamma lattice there are concepts with an empty intent (no methods), the only task of those concepts is to understand how to create composite traits.
- As a last step, we need to introduce those interfaces to recover the polymorphism induced by the original hierarchy (under the hypothesis that we do not introduce any form of inheritance but the interface-based one).

For our case study, we can then analyze the gamma lattice with the goal of individuating the classes, the traits and the interfaces of the refactored hierarchy:

- The 19 concepts that have only one class in their extent will correspond to the concrete classes, implementing all and only the methods in the respective intent; note that we will re-obtain all the classes of the original hierarchy (with a different set of methods).
- The 10 concepts with more than one class in their extent originate 10 traits, shown in Figure 8 (left). We named them T1,...,T10.
- By analyzing the subconcept-superconcept relations in the gamma lattice, we single out the composite traits, shown in Figure 8 (center). Note that the intermediate step of building composite traits is an optional one, that is, the trait-based hierarchy can be build directly either starting from the atomic traits, or by composing such atomic traits in other compositions. Our strategy is based on the idea of composing two traits in a single trait any time there is more than one composite class or trait that uses

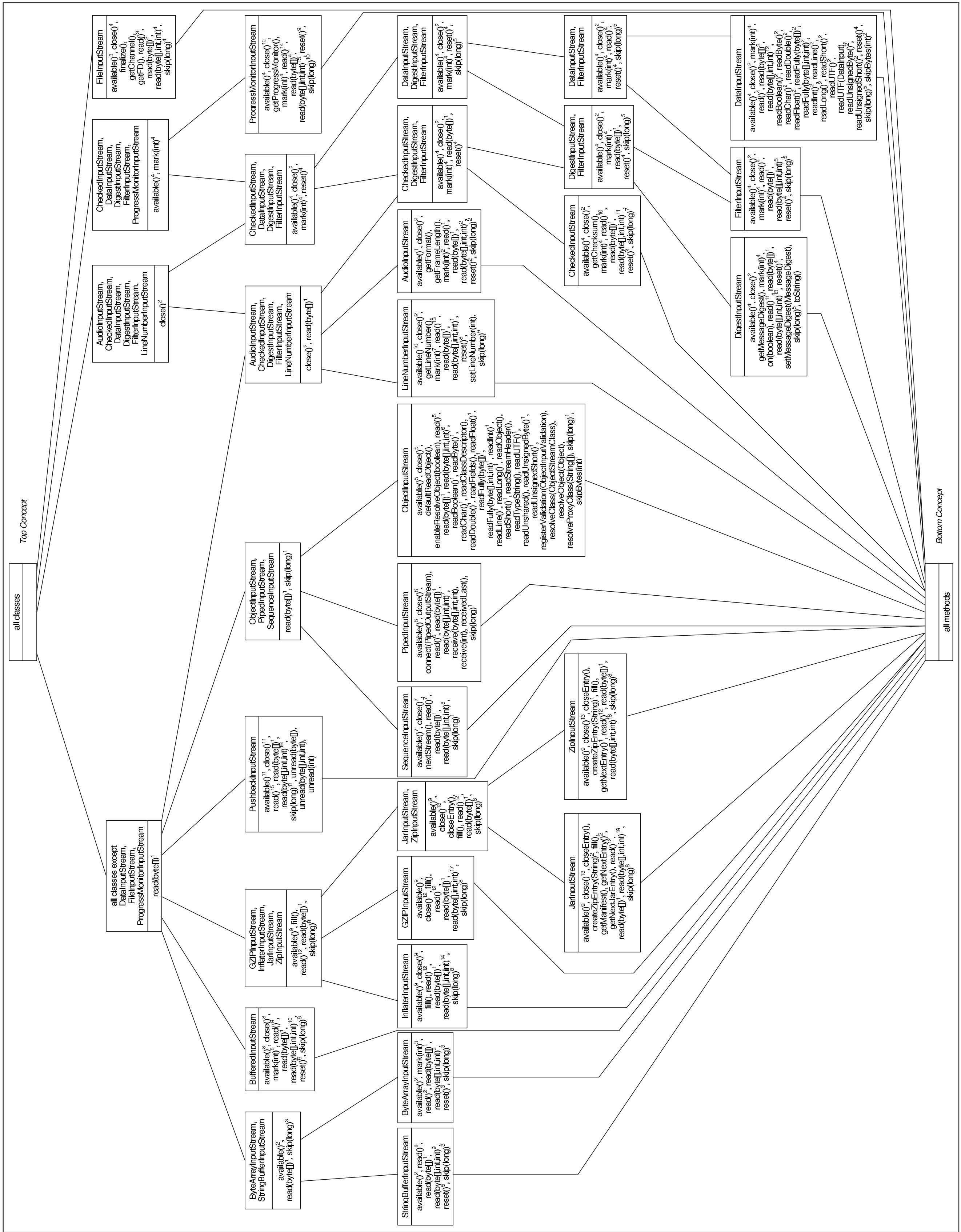

Figure 7: The concept lattice for the Java input stream hierarchy.

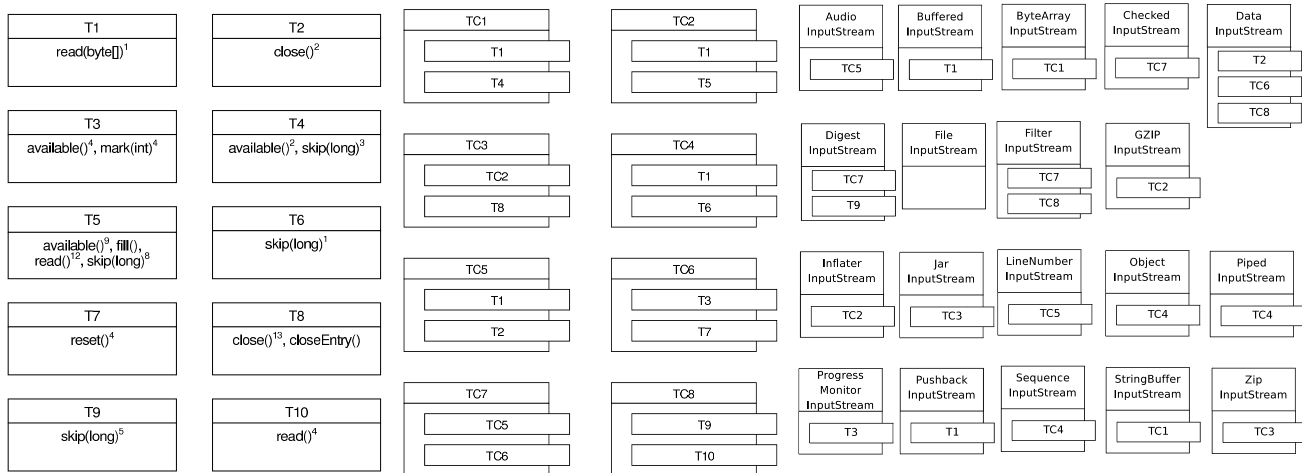

Figure 8: Traits, composite traits and classes for the Java input stream hierarchy.

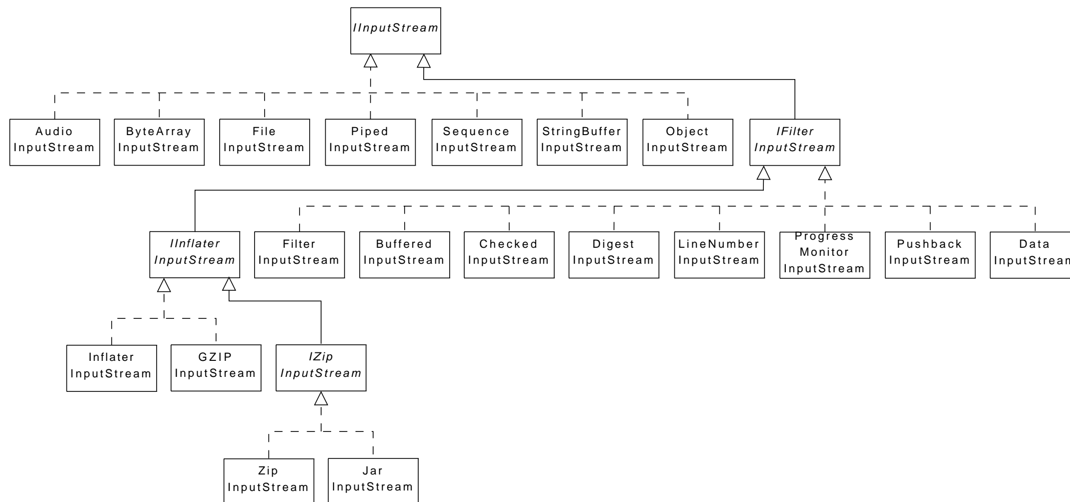

Figure 9: Interfaces for the Java input stream hierarchy.

that particular pair of traits (see Figure 8: trait TC3 is composed with the atomic traits T8 and the composite trait TC2, trait TC7 is composed with the composite traits TC5 and TC6).

- The 19 classes are composed by traits as shown in Figure 8 (right). A special case is class `FileInputStream`, that does not use any trait, because all the methods it needs are not shared by any other classes, therefore they can be implemented directly in the class itself.
- Finally, for restoring the original polymorphism, we organize the classes of Figure 8 (right) in an interface hierarchy, shown in Figure 9. In particular, we introduced an interface for each class that in the original hierarchy has some subclasses. Firstly, we introduced an interface `IInputStream` that declares all the methods declared/implemented in class `InputStream`; this interface is extended appropriately by `IFilterInputStream`, implemented by class `FilterInputStream`, together with all its direct subclasses; and so on. Note that we obtain a refactoring that restores the original polymorphism, but that re-introduces some of the problems that we tried to eliminate. In fact, for example, the methods `mark` and `reset` of `InputStream` must be declared in `IInputStream`, therefore the subclasses that do not support them must implement them as canceled.

5. CONCLUSIONS

In this paper, we proposed a refactoring technique to transform a single-inheritance based hierarchy into a trait based hierarchy, with the aim of improving code reuse. Our proposal is an adaptation to Java of the FCA approach of [19], originally introduced for Smalltalk, which is untyped, and here adapted for a typed setting. The FCA approach was also exploited for identifying modules in legacy code written in C++ in [30].

Our case study is the Java input stream library, but the same approach can be applied to all the main Java libraries, notably the collection and the networking libraries. The obtained results show that our case study may benefit from a trait based refactoring, as most of the flaws of the original version of the library, described in Section 2, were removed. In future work we will then measure how good the refactored output is, from a more quantitative point of view (for instance, the trait granularity), possibly by comparing it with results from manual techniques, as it was done in [11] for

the Smalltalk stream library, that was redesigned from scratch.

We focused on the purpose of designing a trait extraction technique, without choosing any special "Java with traits" as target. The output of our procedure is a lattice of traits that can be seen as a possible refactorization of a classical Java hierarchy, equipped with a hierarchy of interfaces, which is the minimum addition in order not to lose the polymorphism of the original hierarchy. The new trait based hierarchy can be then refined further, with the aid of other techniques, and/or complemented with other programming constructs. We also considered an alternative solution to the "trait + interface" one we adopted in the end, that is, a "trait + interface + abstract class" one, that implies having both trait composition and single inheritance. By using abstract classes, some of the traits individuated in Section 4 (see Figure 8) may be implemented as abstract classes (and possibly refactored in traits in the second phase of the approach); for instance, the root of the hierarchy, the abstract class `InputStream`, would be preserved. Similar considerations to the ones we made about whether to include the methods `mark` and `reset` in the interface `IInputStream` must be taken in account also for the corresponding abstract class. The main advantage of having traits and interfaces only is that we gain in simplicity, while having a form of single inheritance that complements composition might be appealing to programmers used to work with implementation hierarchies.

In the original proposal [19], trait composition is complementary to single inheritance. Note that the main difference between the untyped Smalltalk setting and the typed Java setting is that, for the latter, it is necessary to introduce a form of inheritance among classes along with trait composition in order to maintain the polymorphism. Nevertheless, it is not mandatory to reintroduce class inheritance, but it is enough to introduce an appropriate interface hierarchy. However, from the point of view of having backward compatibility of the refactored library towards old client code, for both settings it would be necessary to re-introduce all subclass-superclass relations that were present originally. This is an important issue and it will be developed in future work.

At the moment, we do not have a clear opinion yet on which would be the best "Java with traits", but we hope that this work would help to give some ideas to reflect upon.

A complementary work is to show formally that the proposed technique produces a hierarchy which is conservative with respect to the semantics of the original one. This will be also the subject of future work, but we believe that it is a straightforward (even still interesting) adaptation of the *flattening* property of [12, 24].

As mentioned earlier, we implemented a tool that performs most of the steps presented in Section 4, in particular all the ones of the first phase, but the bytecode-based detection of duplicated methods.[3] Step 1, that is, the detection of canceled methods, has been semi-automatized, since it is the programmer that has the last word on which among all the syntactically detected methods are actually to be considered as canceled. The tool is an Eclipse plug-in and it is freely available at `http://fcajava.sf.net`.

Acknowledgments. The authors wish to thank Gabriela Arévalo, Damien Cassou, Stéphane Ducasse, and Adrian Kuhn for pointing out alternative, efficient algorithms to perform FCA, and Sara Capecchi, Stéphane Ducasse, and the anonymous referees for useful advice on how to improve the paper.

6. REFERENCES

[1] The Fortress language specification. http://research.sun.com/projects/plrg/fortress.pdf.

[2] D. Ancona, G. Lagorio, and E. Zucca. Jam - A Smooth Extension of Java with Mixins. In *ECOOP 2000*, number 1850 in LNCS, pages 145–178, 2000.

[3] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Squeak by Example*. Square Bracket Associates, 2007.

[4] A. P. Black and N. Schärli. Traits: Tools and methodology. In *Proceedings ICSE 2004*, pages 676–686, 2004.

[5] A. P. Black, N. Schärli, and S. Ducasse. Applying traits to the Smalltalk collection classes. *ACM SIGPLAN Notices*, 38(11):47–64, 2003.

[6] V. Bono, F. Damiani, and E. Giachino. Separating Type, Behavior, and State to Achieve Very Fine-grained Reuse. In *Electronic proceedings of FTfJP'07 (http://www.cs.ru.nl/ftfjp/)*, 2007.

[7] V. Bono, F. Damiani, and E. Giachino. On traits and types in a Java-like setting. In *IFIP TCS 2008 (Track B)*. Springer, 2008. Availble at the url http://www.di.unito.it/~damiani/papers/tcs08B.pdf.

[8] G. Bracha. *The programming language JIGSAW: mixins, modularity and multiple inheritance*. PhD thesis, Department of Comp. Sci., Salt Lake City, UT, USA, 1992.

[9] G. Bracha and W. Cook. Mixin-Based Inheritance. *ACM SIGPLAN Notices*, 25(10):303–311, Oct. 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).

[10] G. Bracha and D. Griswold. Extending Smalltalk with mixins. In *OOPSLA96 Workshop on Extending the Smalltalk Language*, 1996. http://www.javasoft.com/people/gbracha/mwp.html.

[11] D. Cassou, S. Ducasse, and R. Wuyts. Redesigning with traits: the Nile stream trait-based library. In *ICDL*, pages 50–75, 2007.

[12] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2):331–388, 2006.

[13] S. Ducasse, N. Schärli, and R. Wuyts. Uniform and safe metaclass composition. *Computer Languages, Systems and Structures*, 31(3–4):143–164, 2005.

[14] K. Fisher and J. Reppy. Statically typed traits. Technical Report TR-2003-13, University of Chicago, Department of Computer Science, Dec. 2003.

[15] K. Fisher and J. Reppy. A typed calculus of traits. In *Electronic proceedings of FOOL 2004*, 2004.

[16] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. POPL '98*, pages 171–183, 1998.

[17] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1999.

[18] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Conference Proceedings of OOPSLA '97, Atlanta*, volume 32(10) of *ACM SIGPLAN Notices*, pages 318–326. ACM, 1997.

[19] A. Lienhard, S. Ducasse, and G. Arévalo. Identifying traits with formal concept analysis. In *Proc. 20th Conference on Automated Software Engineering (ASE'05)*, pages 66–75. IEEE Computer Society, 2005.

[20] C. Lindig. *Fast Concept Analysis*. PhD thesis, Harvard University, Division of Engineering and Applied Sciences, Cambridge, Massachusetts, 2002.

[21] L. Liquori and A. Spiwack. Feathertrait: A modest extension of Featherweight Java. *ACM TOPLAS*, 2008.

[22] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1991.

[23] M. Naddeo. Un possibile approccio alla soluzione di alcuni problemi legati all'ereditarietà singola nei linguaggi object-oriented. Laurea triennale in informatica, Università degli Studi di Torino, 2008.

[24] O. Nierstrasz, S. Ducasse, and N. Schärli. Flattening traits. *JOT (www.jot.fm)*, 5(4):129–148, 2006.

[25] M. Odersky. The Scala Language Specification, version 2.4. Technical report, Programming Methods Laboratory, EPFL, Switzerland, 2007.

[26] P. J. Quitslund, R. Murphy-Hill, and A. P. Black. Supporting Java traits in Eclipse. In *OOPSLA workshop on Eclipse Technology eXchange - ETX 2004*, pages 37–41. ACM, 2004.

[27] J. Reppy and A. Turon. A foundation for trait-based metaprogramming. In *Electronic proceedings of FOOL/WOOD 2006*, 2006.

[28] J. Reppy and A. Turon. Metaprogramming with traits. In *ECOOP 2007*, volume 4609 of *LNCS*, pages 373–398. Springer, 2007.

[29] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *ECOOP 2003*, volume 2743 of *LNCS*, pages 248–274. Springer, 2003.

[30] M. Siff and T. Reps. Identifying modules via concept analysis. *IEEE Trans. Softw. Eng.*, 25(6):749–768, 1999.

[31] C. Smith and S. Drossopoulou. *Chai*: Traits for Java-like languages. In *ECOOP'05*, LNCS 3586, pages 453–478. Springer, 2005.

[32] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[3] The tool does not create in output the graphical form of the incidence table and of the lattices, but it generates all the necessary information to build them.