Migrating inheritance-based applications into components

Nicolás Passerini, Gabriela Arévalo

Universidad Nacional de Quilmes, Bernal, Argentina
npasserini@gmail.com, garevalo@unq.edu.ar

Abstract. Inheritance is one of the most important object-oriented mechanisms, allowing code reuse and a conceptual hierachical modelling of a given domain. However, as a white-box reuse mechanism, it introduces hidden (implicit) coupling between classes in a hierarchy. This problem makes understanding and maintenance difficult to grasp.
Actual approaches (such as Gamma's) propose *object composition* over *class inheritance* as reuse mechanism to obtain better object-oriented design. However the migration of a class-based application (or just some software artifacts) from inheritance to composition paradigms is not trivial. To our knowledge, there are only a few approaches that can deal with this problem, but there is no formal language-independent analysis of a transformation algorithm that guarantees the exact same behavior of a system once the migration strategies have been applied. Thus, this paper presents some initial results in designing a *refactoring* approach for a class hierarchy by analyzing the dependencies between the classes involved in the inheritance relationship in a class-based system, and how they can be transformed to obtain a better structure of the class hierarchy, focusing on getting components with *offered-* and *required-services* in component-based system. Based on this study, our main goal in this approach is to develop a (semi)automatic language-independent algorithm using refactoring-based strategies which allows the user to transform an inheritance relationship between a set of classes into an association between two independent *components* with well defined *interfaces*.

1 Introduction

All software systems are exposed to changes during their lifecycle [6]. Many companies are facing the fact that most of the changes cannot be predicted (unanticipated changes), because they are driven by the market or emerging trends and technologies. Various approaches have been proposed to solve this problem, such as *component-based software development* (CBSD) [7] that proposes to build applications out of validated and substitutable and reusable components. This component-based view of software is also one of the keys to the transition to the *service-oriented* paradigm (SOP) and *software as a service* approaches. Even when the CBSD and SOP approaches are useful in building applications, most of existing applications are implemented in class-based languages, where the main building mechanisms are the inheritance and the polymorphism. Knowing all the advantages that CBSD and SOP approaches have with components as black-box mechanism, we believe that it is worth to develop migration strategies that allow

the developers to transform the inheritance-based application into a component-based one. However, the identification of the components in an existing object-oriented application is not trivial, because the object-oriented building mechanisms generate strong *hidden* coupling between the software entities in a system. Even with well-designed applications, these implicit mechanisms make the understanding of the original structure (module, packages, classes) difficult to maintain, and their transformation to support evolution into component-based approaches results as a hard task.

The goal of our research is the design of an approach to the development of refactoring-based strategies to remodularize and extract software entities focusing on migrating an application from an inheritance to a component-based system. To achieve this goal, in this paper we present some initial results in the analysis of dependencies between classes in a system, and our initial steps to design an algorithm that (semi-)automatically perform the extraction of the entities from the source-code of object-oriented systems.

The paper is structured as: section 2 describes our approach for the refactoring, first in general and then subsequent sections 2.2 to 2.6 detail each step of the proposed algorithm. Section 3 describes related works and section 4 presents our conclusions.

2 Inheritance to Composition: Our Approach

Inheritance is one of the most important object-oriented mechanisms, allowing code reuse and a conceptual hierachical modelling of a given domain. Inheritance combined with other object-oriented mechanisms, such as overriding, method cancellations, encapsulation and polymorphism can make class hierarchies difficult to understand, because unexpected and implicit behavior based-dependencies can appear. One of these difficulties is the use of inheritance for implementation reuse (subclassing) possibly violating subclassing contracts from structural and behavioral viewpoints [1]. To illustrate our concepts, we use an example of some classes in `Collection` class hierarchy developed in Pharo[1]. In Figure 1, the class `Dictionary` inherits from `HashedCollection`. `Dictionary` elements can only be removed by their key values, so the method `#remove:` inherited from `HashedCollection` is cancelled in the class. From a structural viewpoint, inheritance is used with implementation purposes, and it forces the developers to cancel inherited methods. From the behavioral viewpoint, they are incomparable because this hierarchy does not repect the Liskov-Substitution Principle [5]. In order to solve this *code smell*, Fowler proposes to use delegation instead [2]. The proposed refactoring strategy is described as follows:

- Implement `Dictionary` as a direct subclass of `Object`[2].
- Add an instance variable named `delegate` in the modified class `Dictionary` and initialize it as a `HashedCollection` instance.
- Change each method defined in the modified class `Dictionary`, replacing self- or super-sends with messages to `delegate`.

[1] http://www.pharo-project.org

[2] Pharo is a single-inheritance-based system. Object is the root class of the system

– For each method defined in all superclasses of the original `Dictionary` (i. e. `HashedCollection` or `Collection`) used by a client class and not redefined in `Dictionary`, add a simple *delegating* method.

Figure 4 shows the refactored class hierarchy with the *refactored class Dictionary*.

Even though the refactoring proposed by Fowler is a useful guide, the algorithm is difficult to implement because we have to take care of the implicit dependencies determined by the message calls between classes to ensure that the resulting code maintains the exact same behaviour as the original implementation. Following we detail our algorithm and we analyze all the issues related to the classes and methods that must be refactored.

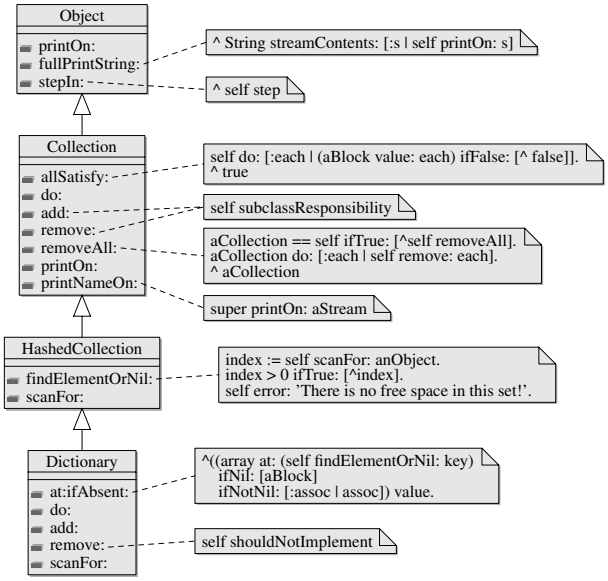

Fig. 1. Inheritance to Delegation Refactoring – Original class Hierarchy

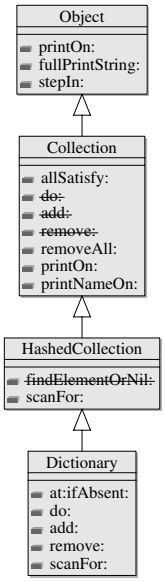

Fig. 2. Initial Code Base.

An important remark is that even when our algorithm is based on Fowler's work, our approach goes a step further. The refactoring operations of changing the methods' definition of the *refactored* class implies more than simply using *delegation* mechanism, and this is explained in detail in last steps of the algorithm.

2.1 Refactoring Algorithm: Outline

The goal of our approach is to develop an algorithm to extract a class from a class hierarchy, preserving the exact original behavior. To illustrate our algorithm we use a small example to explain the main concerns to be affected. Thus, our algorithm analyzes the

dependencies between the classes in the hierarchy, focusing on the *self* and *super* sends that invoke methods defined in a different class than the one where the message send is done. When we refactor the class hierarchy, we will modify the lookup chain of these message sends, so we have to modify each method definition in order to ensure that the same code as in the original implementation. Just to mention some problems, one problematic case is that when we remove the `Dictionary` and reimplement it as a direct subclass of `Object`, we are *virtually removing* `HashedCollection` and `Collection` (intermediate superclasses between `Object` and `Dictionary`). Another problematic case is when methods defined in `Object` and inherited by `Dictionary` invoke, via self sends, methods defined in the *removed* superclasses. Finally, an inherited method defined in `HashedCollection` could invoke, via a self send, an overriden method in the subclass `Dictionary`.

Summarizing our approach, we need first to identify different methods that can be potential problems to preserve the original behavior and then specify which are the different steps that the algorithm should perform.

Regarding how to organize the code analysis, we have to calculate the set of all methods that could be invoked as a result of a message sent to a `Dictionary` object, both from an external client or internally via self- or super-sends. This code base is divided in three groups. The first group *own-group* contains all the methods defined in the `Dictionary` class. The second group named *inherited-group* contains the methods defined in the superclasses of `Dictionary` excluding `Object`, in our example the methods belong to the classes `HashedCollection` and `Collection`. Finally the methods in the code base coming from `Object` (and its superclases, if any) will conform the *object-group*.

The goal of the algorithm is to identify all the refactoring operations that the developer should implement to get the changes done, this means that the algorithm does not perform any changes in the code when running. We define an algorithm in 5 steps.

- **Step 1:** We analyze the code and we calculate the initial code base.
- **Step 2:** We will clean-up the code base, removing cancelled methods and other typical problems arised from the inadequate use of inheritance.
- **Step 3:** We decide the needed delegating methods to be added to the target class. The final two steps deal with the modifications on the self- and super-sends in the code base.
- **Step 4:** We change self- and super-sends in the own-group, using the `delegate` where needed.
- **Step 5:** We analyze self-messages in the inherited-group.

2.2 Initial Code Base

In this step we filter the methods of the to-be-refactored hierarchy, keeping only those that have to be analyzed in the next steps of the algorithm. We discard all the methods that could never called as response to a message sent to an instance of the to-be-refactored class (in our example, `Dictionary`). This is performed in three stages. In the first stage we identify the set of methods defined in the to-be-refactored class or any

of its superclasses. In the second stage we discard all the overriden methods. When a method is defined in more than one class in the hierarchy, we keep only the one defined in the most specific class. When executing this action on `Dictionary` we found 598 methods (this could change in different versions of Pharo).

The first two stages identify the set of methods that can be called by a client class of the `Dictionary` or as result of a self send. We mean by a client class, one that is not contained in the same class hierarchy of `Dictionary`. In the third stage we identify the methods that can be called as result of a super send. We search in the methods containing super-sends in their bodies, then lookup the method that is invoked as response to these messsages sends and add those methods, (that could be previously discarded, because they fulfil previous conditions) to the code base. As these new added methods could also contain super-sends, we have to repeat recursively this action until no new methods are found. In the `Dictionary` hierarchy we found only one example of super-send.

Figure 2 depicts the initial code base. This is only a representation of our working model, no actual code refactoring is done. We have discarded all overriden methods, except `Object≫#printOn:`, which is invoked by a super-send in `#printNameOn:` from class `Collection`

2.3 Clean up the code base

The inadequate use of inheritance makes the class understand messages that are inconsistent with the semantics of the represented domain concept. For example, `HashedCollection` implements method `remove:`, which class `Dictionary` inherits, due to the inheritance relationship. However, this behavior should not be in the `Dictionary`, because dictionaries remove their elements using the keys associated to the stored values. Thus, elements are removed from a `Dictionary` using the method `removeKey:`. Before performing the refactoring, we must detect those unwanted methods and remove them from the code base.

We have detected four kinds of unwanted messages. The most explicit kind of unwanted methods are the *cancelled methods*[3]. The second kind of unwanted methods are the abstract methods that were not overriden in the subclasses. This situation could also be considered as a mistake of the developer of the to-be-refactored class, but we prefer to assume that the initial code is correct, and that the method was left unimplemented intentionally. Therefore we consider it as unwanted. In a dynamically typed language as Smalltalk, the programmer is not forced to make abstract methods as concrete ones in the subclasses, hence it is possibly to find, in an abstract class, messages sends which have no concrete implementation. If the (concrete) subclass does not provide an implementation for this messages, we consider them as unwanted too[4]. Finally, we have to lookup the code base for methods that call any discarded message, and discard them

[3] In Smalltalk, the method cancellation is implemented using the message `self shouldNotImplement` as the only message sent in the body of the cancelled method

[4] This methods will not be found in the code base, because there is no implementation of them, still they have to be taken into account in order to discard the senders of this messages.

too. Again, as this final action discarded new methdos, the senders of these messages have to be removed too, repeating this action until no new methods are found.

Table 1 shows the discarded methods found in the class `Dictionary`. A total of 10 methods were found, including two of them that were selected twice because they fullfil several conditions. For clarity, only a subset of these methods has been included in the class diagrams of the Figure 3.

Cancelled methods	2	`#remove:ifAbsent:` `#remove:`
Abstract non-overriden methods	2	`#remove:ifAbsent:` `#remove:`
Sent non-implemented methods	2	`#readFrom:` `#step`
Methods sending improper messages	6	`#removeAllFoundIn:` `#removeAll:` `#removeAllSuchThat:` `#stepAt:in:` `#stepIn:` `#readFromString:`

Table 1. Discarded methods in `Dictionary` hierarchy

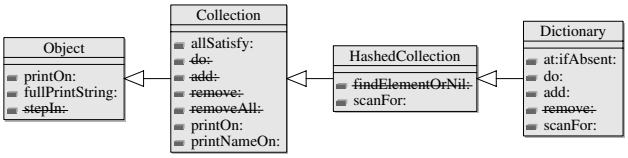

Fig. 3. Final code base with discarded methods removed.

2.4 Add needed delegating methods

In the original implementation, the to-be-refactored class inherits all the methods of its superclasses. But the refactored class has now `Object` as it superclass and then it will not understand all the messages implemented in its former superclasses and not overriden in the original class itself. In our specific case, the refactored class `Dictionary` will not understand the inherited methods defined in classes `Collection` and `HashedCollection`. Therefore, we have to analyze each of these messages and decide if we need the refactored class continue understanding them to keep the polymorphism of the class.

We have identified several strategies to deal with this problem. In a statically typed language, this could be achieved using of any of the explicitly defined types of the refactored class. Even when the explicit type information is an important tool to compute the required messages more accurately, but it also is a limitation if the refactored class has to be kept *polymorphic* to any other class in the system. Kegel [3] provides a solution of this problem in Java . He looks for an existing defined common `interface` or superclass between refactored classes in order to allow polymorphism. In dynamically typed

languages, such as any language built on top of Smalltalk polymorphism is achieved just by understanding the same messages, so we can avoid some of these problems. Nevertheless, the lack of explicit type information makes an automatic computation of the needed interface for the refactored class more difficult to get: we should compute it via a type inference algorithm, and this is the first strategy we identify. Another possible strategy is to provide a user interface, which lets the programmer to decide which of the available methods have to be included.

In our approach, we choose to take all of the methods in the superclass as part of the new public interface of the *refactored* class, and provide *delegating* methods for all of them. When we refer to *delegating methods*, we mean by a method with the same name which sends the message (with the same name) to the created `delegate` instance variable. Figure 4 shows the structure of the recent created methods in the refactored `Dictionary`. This solution provides the simplest refactoring and minimizes its behavioral change, which is a desired feature in an automatic refactoring. Our algorithm is implemented using the last described strategy, but can be adapted to integrate new ones (as we described previously).

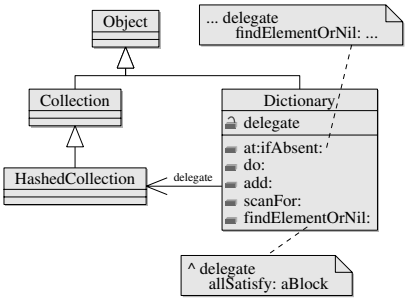

Fig. 4. Modifications to the Dictionary class.

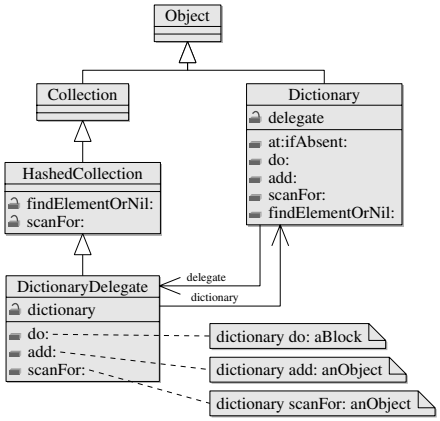

Fig. 5. Final state of the refactor.

2.5 Apply delegations

In this step we analyze in detail all the methods that were selected in Section 2.3. We analyze those self sends in the *own-* or *object-group* that invoke methods defined in the *inherited-group*. In our example, this means that we analyze any method that belongs to the `Object` or to the `Dictionary` that makes a *self send* to a method defined in any of the superclass `Collection` and `HashedCollection`. The objective is to change the receiver of these messages sends, using instance variable `delegate` instead of *self* or *super*. This analysis could become complex in a large hierarchy chain if we

do not have correct analysis tools. Our strategy is to compute the *intergroup message sends*, of the whole set of methods that could be invoked of the to-be-refactored class. We consider a self- or super-sends as member of the intergroup when the message yields the invocation of a method belonging to a different group than the method where the message-send is done. For example, Figure 1 shows that the message-send `self findElementOrNil:` in `Dictionary≫#at:ifAbsent:` belongs to the *own-group* and yields the invoked method `HashedCollection≫#findElementOrNil:`, which belongs to the *inherited-group*.

With the computation of message-sends belonging to the intergroup , we have to analyze separately the message-sends of the own-group and those ones of the object-group. In the object-group we have two limitations. Firstly, we do not want to change code in `Object`. Second, we do not have access to the `delegate`. Then, for each self-send originated in the methods of the *object-group*, we have to provide a delegating method in the refactored class, even if it has been discarded in the previous step. In the *own-group* we can also take advantage of the selected delegating methods, if a method belonging to the *intergroup* uses a method that was selected to create a delegating method in the refactored class, then no change is needed. Finally, in the case of super-sends in the own-group[5]. we choose to delegate them always. Table 2 summarizes the actions of this step, while Figure 4 shows the resulting methods. As an example, method `#at:ifAbsent` a former reference to `self findElementOrNil:` has been changed to `delegate findElementOrNil:`

Group	Receiver	Should be delegated?	Example
own	self	When delegating method is not created	`self findElementOrNil:` in `Dictionary≫#at:put:`
own	super	Allways	no examples in `Dictionary` hierarchy.
object	self	Never. Delegating method must be created.	`self printOn:` in `Object≫#fullPrintString`

Table 2. Actions taken for intergroup messages

2.6 Self-messages in the inherited group

A self-send of a message in a method of the *inherited-group* could yield an invocation of a method in the *own-group*. This means that a method of the classes `Collection` or `HashedCollection` could call a method defined in the `Dictionary`. This is a very useful property of inheritance-based systems, enabling design patterns such as *Template Method*. This mechanism also makes the communication between the code in the superclass and its subclasses bi-directional.

[5] Super-send of the message in the object-group cannot be intergroup

A complete replacement of class hierarchy with a single delegation can only be achieved in languages which provide a *delegation mechanism* where the metavariable *self* allways points to the delegator and not to the method receiver [4]. This kind of delegation is frequently found in prototype-based object-oriented languages such as Self[8], but rather uncommon in class-based languages. In order to make our strategy as language-independent as possible, we prefer to avoid the usage of this kind of features. To achieve it, we need to provide a *counter-delegation* mechanism, that will enable the code in the *inherited-group* to continue forwarding messages to the *own-group*, as in the original inheritance-based implementation.

The implementation of counter-delegation is rather tricky if we keep our objective of avoid changing the superclasses. A common strategy is to create a *counter-delegating class*. The new class will have the same superclass as our original class, allowing to override its methods, modifying its behaviour without changing it. It will also have a `delegator` field, which points to the main object, an instance of the refactored class. In this strategy, the `delegate` field of the refactored class will not point to an instance of its former superclass, but to an instance to the new counter-delegating class. By doing so, the two objects will know each other, enabling the bi-directional communication as needed. Figure 5 shows the refactored classes.

It is worth mentioning of introducing again inheritance is a conflict with our original goal. The new generated subclass is much simpler than the original one, with only counter-delegating methods. The main objective of the refactoring is not to avoid the use of inheritance at all, but to decouple the code in the original class from the code in its superclasses, providing a clean view of the dependencies between them. In languages with *nested classes* such as Java, the counter-delegating class could be encapsulated into the main refactored class by making it private[3]. If language-independence were not an issue, a broader range of solutions become available if we take advantage of features like full delegation (as found in prototype-based languages), nested classes, multiple inheritance or traits. In this work, we stated as an objective to develop a language-independent refactoring, so we need to avoid using language-specific features.

3 Related Work

Due to space limitations, we just mention the most recent work that is close to our approach. Kegel and Steiman [3] define an inheritance-to-delegation refactoring in Java, making a distinction between *delegation*, which allows open recursion and *forwarding*, which not; the latter being more simple but applicable in fewer cases. The open recursion problem is solved by creating a reverse-delegating subclass, other solutions (like trait-based) to the open are not considered. The work is based on a strongly-typed environment with explicitly-typed variables; therefore direct knowledge of the refactored class interface is available. On the down side, the strongly-typed language imposes some restrictions on the aplicability of the refactoring, as the refactored class will could not be used polimorphically with its former superclass and siblings, unless an explicit supertype is given by implementing a shared `interface`. According to their work, type inference is needed to make this refactoring. However, the typing rules of Java

impose the presence of many *dead methods*, which could be avoided in a dinamically-typed language as Smalltalk. Also, *cancelled methods* are not considered at all, nor the refactoring where the original superclass is abstract.

4 Conclusions and Future Work

In this paper we have shown the initial steps of our research in the design *refactoring* strategies for remodularization of the object-oriented applications focusing on migration of an inheritance to a component-based application. We have mainly focused on the migration of a class in an existing class hierarchy to improve its definition and avoid *subclassing* problems. We have observed several improvements that can be done in our approach. We just mention some of them. We can use type inference to know which of the inherited methods should be effectively be invoked by a client class, or we can also use traits to improve the class definition.

References

1. Arévalo, G., Ducasse, S., Gordillo, S.E., Nierstrasz, O.: Generating a catalog of unanticipated schemas in class hierarchies using formal concept analysis. Information & Software Technology 52(11), 1167–1187 (2010)
2. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison Wesley (1999)
3. Kegel, H., Steimann, F.: Systematically refactoring inheritance to delegation in java. In: Proceedings of the 30th international conference on Software engineering. pp. 431–440. ICSE '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1368088.1368147
4. Lieberman, H.: Using prototypical objects to implement shared behavior in object oriented systems. In: Proceedings OOPSLA '86, ACM SIGPLAN Notices. vol. 21, pp. 214–223 (Nov 1986), http://web.media.mit.edu/~lieber/Lieberary/OOP/Delegation/Delegation.htmlhttp://reference.kfupm.edu.sa/content/u/s/using_prototypical_objects_to_implement__76339.pdf
5. Liskov, B., Wing, J.M.: A new definition of the subtype relation. In: Nierstrasz, O. (ed.) Proceedings ECOOP '93. LNCS, vol. 707, pp. 118–141. Springer-Verlag, Kaiserslautern, Germany (Jul 1993), http://link.springer.de/link/service/series/0558/tocs/t0707.htm
6. Parnas, D.L.: Software aging. In: Proceedings of the16th ICSE '94. pp. 279–287. IEEE Computer Society, Los Alamitos CA (1994)
7. Szyperski, C.A.: Component Software. Addison Wesley (1998)
8. Ungar, D., Smith, R.B.: Self: The power of simplicity. In: Proceedings OOPSLA '87, ACM SIGPLAN Notices. vol. 22, pp. 227–242 (Dec 1987)