Towards Automatic Decoration

Tomoyuki Aotani
Tokyo Institute of Technology
aotani@c.titech.ac.jp

Tetsuo Kamina
Ritsumeikan University
kamina@cs.ritsumei.ac.jp

Abstract

It is important from the view point of separation of concerns to separate optional features from core features of classes. The decorator pattern is a major technique to achieve this in single inheritance object-oriented programming languages such as Java. There are also more advanced techniques for modularity such as multiple inheritance, mixins, traits, and incomplete objects. This direction is helpful for implementers of classes in the sense that it helps to avoid code duplication. It however makes the use of the classes complicated because one has to pick up and compose the modules that provides optional features with the modules that provides core features manually. We propose decorators as a solution to the problem. Decorators are a simple extension of mixins. One of the important differences of decorators from mixins is automatic composition: decorators are intended to be composed with classes automatically by inferring the set of decorators from the use of objects, which we call decoration inference. This paper sketches and demonstrates decorators and decoration inference through simple and small examples. It also gives and discusses issues to realize decorators.

1. Introduction

It is natural today for better modularity to separate optional features from core features when we design classes. For example, the java.io package in Java uses the decorator pattern (Gamma et al. 1995) instead of inheritance to provide input and output streams (e.g. `FileInputStream`) with a variety of features such as handling character encodings (e.g. `InputStreamReader`), data compression (e.g. `ZipInputStream`), buffering (e.g. `BufferedInputStream`) and encryption (e.g. `CipherInputStream`). There are also more advanced techniques for modularity such as multiple inheritance, mixins (Bracha and Cook 1990), traits (Schärli et al. 2003), and incomplete objects (Bettini et al. 2011).

These fine-grained modules make the composition complicated. We have to compose them with core classes or objects manually to use optional features, even though the necessary modules are clear in many cases. Moreover, because different modules have different dependencies with respect to composition, it is sometimes painful to change code to use a different set of optional features. For example, it is common in Java to read the content of a text file by line by (1) creating a `FileInputStream` object, (2) wrapping it with `InputStreamReader` and `BufferedReader` in the order, and (3) calling the `readLine` method to the `BufferedReader` object.[1] To read the context of a binary file, on the other hand, we usually (1) create a `FileInputStream` object as the previous example, (2) wrap it with `BufferedInputStream` and `DataInputStream` in the order, and (3) call the `readByte` and similar methods to the `DataInputStream` object. Therefore, if we change a program that reads a text file to the one that reads a binary file, we have to change not only the code to read the content but also the code for composition.

We propose *decorator*s as a solution to the problem. Decorators are similar to mixins but they are composed *implicitly* with classes and other mixins. This implicit composition, which we call *decoration*, is performed when each object is created. Which decorators are composed depends on the use of each object. The implicit composition makes programmers free from composing fine-grained modules manually. For example, all that one has to do to read the context of a text file by line is to create a `FileInputStream` object and then call the `readLine` method to it if `InputStreamReader` and `BufferedReader` are decorators. Similarly, to read the content of a binary file, one creates a `FileInputStream` object and then calls the `readByte` method to it if `BufferedInputStream` and `DataInputStream` are decorators.

Our notable feature is *decoration inference* that collects the set of necessary decorations automatically by analyzing how the objects are used. Decoration inference is similar to type inference for first class messages (Müller and Nishimura 2000) and records (Rémy 1994) in ML families of languages. Decoration runs at runtime but decoration inference runs before runtime.

In this paper, we sketch and demonstrate decorators and decoration inference. We also give and discuss issues to realize decorators. For easy understanding, we use a hypothetical language that extends Scala with decorators.

[1] The `Files` class available as of Java7 provides the `readAllLines` method to read all lines from a file by just specifying the path to a file and the character encoding. It however cannot be a solution essentially.

```
abstract class InputStream{ abstract def read(); }
class FileInputStream(f:File) extends InputStream{...}
trait Reader{
  abstract def reads(cbuf:Array[Char],
                     offset:Int, length:Int):Int;}
trait InputStreamReader extends Reader
                              decorates InputStream{
  def setEncoding(cs:Charset):Unit={...}
  override def reads(cbuf:Array[Char],
                     offs:Int, len:Int):Int={...}}
trait BufferedReader decorates Reader{
  def setBufferSize(size:Int):Unit={...}
  def readLine():String = {...}}
```

Listing 1. Reader and its decorators

```
val fr = new FileInputStream(aFile);
val lines : Iterator[String] =
  Iterator.continually(fr.readLine())
          .takeWhile(_ != null);
lines.foreach{println(_)};
fr.close();
```

Listing 2. A client of the Readers

2. Decorators

Decorators are similar to mixins but intended to be composed with other classes implicitly. To this end, each decorator has a decorates clause that specifies the classes, mixins and decorators with which it is composed. We call such classes, mixins and decorators *decoratee*s.

Listing 1 reimplements, using decorators, InputStream, Reader, and their subclasses found in the java.io package. To allow one to create Reader objects from InputStream objects, we declare Reader and InputStream as traits instead of classes.

InputStreamReader adds two methods, setEncodings and reads to InputStream (and its subtype) when that decorator is composed with the decoratees. BufferedReader similarly adds two methods, setBufferSize and readLine to Reader.

One major difference from Scala's trait is the approach to composition. Listing 2 shows a simple client that prints the content of a file line by line. In Scala, composition is always explicit and thus the client code is invalid, because FileInputStream does not declare the readLine method. When writing the code in line 3, one may eventually become aware that, to use readLine, FileInputStream must be composed with InputStreamReader and BufferedReader. Then, one must go back to and reedit line 1. This happens always when one requires additional features.

In our mechanism, on the other hand, composition is implicit; one does not need to write composition. The necessary decorators are automatically *inferred* from the client code. Therefore, the client code in Listing 2 is valid.

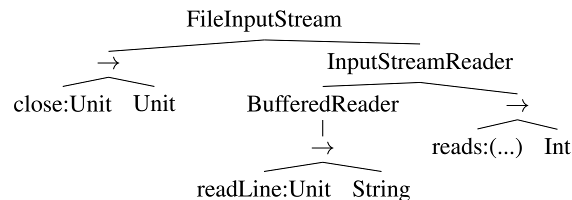

Figure 1. Decorator tree for FileInputStream

3. Decoration inference

Decoration inference finds the set of decorators to be composed by analyzing the use of each objects statically. The inference process consists of two steps, namely constraint building and decorator finding.

The constraint building uses similar technique to the type inference of records in ML family of languages and first-class messages in object-oriented languages. In Listing 2, the objects of FileInputStream are used at two sites. The first use is receiving the call to the method readLine in line 3. The second use is receiving the call to the method close in line 6. Therefore, the decoration inference build a constraint $\texttt{fr} \leq (\texttt{FileInputStream} \wedge \{\texttt{readLine} : \texttt{Unit} \rightarrow \texttt{String}\} \wedge \{\texttt{close} : \texttt{Unit} \rightarrow \texttt{Any}\})$.

To find a suitable set of decorators, we have to have the set of decorators that can be composed with FileInputStream. Figure 1 shows the feature tree for FileInputStream. It is easy to build the tree if we have the full set of decorators. Because decorators specify the target type of decoration, we can know the set of decorators for FileInputStream. Here, it is a singleton that has only InputStreamReader. We also can get the set of decorators for InputStreamReader, which is a singleton that has only BufferedReader.

Finally, we fix the necessary decorators and its composition order by finding the paths to the called methods. From the constraint, we know that the result of the composition must have the readLine and close methods. There is no decorator on the path from FileInputStream to close and are InputStreamReader and BufferedReader on the path from FileInputStream to readLine. We thus conclude that we have to compose InputStreamReader and BufferedReader with FileInputStream in that order.

4. Discussions

There are of course several technical issues to realize decorators. In this section, we show three of the issues.

4.1 Combination with type inference

It is not trivial to support both decoration inference and type inference. Modern statically-typed languages support type inference. It infers types of expressions and variables based on their use in the program. To support decorators in such languages, decoration inference must be compatible with type inference.

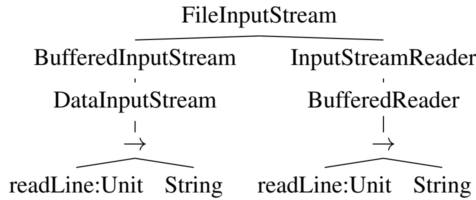

Figure 2. Ambiguous decorator tree for `FileInputStream`

Supporting these two inferences is, however, problematic. Type inference often assumes that expressions are complete, i.e., it simply rejects the expressions and variables if it cannot find any type for them. Decoration inference on the other hand must run on incomplete programs and fix them by adding decorators to the expressions that create objects.

It is not a solution to run decoration inference before type inference. This is because decoration inference depends on the types of the expressions that use the objects whose types are not complete.

4.2 Ambiguity

Decorator inference is not trivial if there are two or more decorators in the decorator tree that provide methods with the same name and type. Minimizing the number of selected decorators is one candidate of the solutions, although of course it does not work fine with all cases.

One example of such cases that are easy to be solved is a method that calls `close` to a `FileInputStream` object but does not call `readLine` to it (i.e., lines 2–5 are omitted from Listing 2) and the `FileInputStream` class and `BufferedReader` decorator provide it. The constraint for the object, say `fr`, is $\texttt{fr} \leq (\texttt{FileInputStream} \wedge \{\texttt{close} : \texttt{Unit} \rightarrow \texttt{Any}\})$ and the decorator tree for `FileInputStream` extends Figure 1 with the `close` method as another child of `BufferedReader`. Naïve decorator inference fails to find a set of decorators because there are two candidates, i.e., the empty set and the set containing `InputStreamReader` and `BufferedReader`. In this case, we can easily choose the empty set by minimizing the number of selected decorators.

There is also, however, an example of problematic cases in practice. Suppose a method that calls only `readLine` to an `FileInputStream` object and the two decorators `BufferedReader` and `DataInputStream`, which is defined as follows[2]:

```
traint BufferedInputStream decorates InputStream{...}
trait DataInputStream decorates BufferedInputStream{
  def readLine():String={...}}
```

In this case, the constraint for the object, say `fr`, is $\texttt{fr} \leq (\texttt{FileInputStream} \wedge \{\texttt{readLine} : \texttt{Unit} \rightarrow \texttt{String}\})$ and the decorator tree for `FileInputStream` is Figure 2. Naïvedecoration inference again fails because there are two candidates, those are, the set of `BufferedInputStream` and `DataInputStream` and the set of `InputStreamReader` and `BufferedReader`. We cannot choose one set because their size are the same.

4.3 Modular decoration over methods

Modular decoration over methods is necessary It is common that a method uses objects created in other methods. We call the former method a *builder method* and latter methods *consumer method*s. Decoration over methods composes decorators with core classes depending on how the builder method use them. We say it is modular if it does not check the body of each method more than once.

The following code demonstrates the advantage. It is a re-implementation of Listing 2. The method `builder` creates a `FileInputStream` object and returns it. The method `consumer` first calls `builder` to get an object and then calls `readLine` and `close` on the object.

```
def builder(){ new FileInputStream("some file"); }
def consumer(){
  val fr = builder(); /* same to Listing.2*/; }
```

Builder methods have to compose different sets of decorators depending on consumer methods to support the modular decoration between methods. For example, the method `builder` in the above code must compose `InputStreamReader` and `BufferedReader` with `FileInputStream` as requested by the method `consumer`.

[2] There must be a discussion on how to design decorators. We suppose that `DataInputStream` should decorate `BufferedInputStream` instead of `InputStream` because use of `BufferedInputStream` is always recommended.

Acknowledgments

We would like to thank the anonymous reviewers of the NOOL 2016 workshop for their comments and suggestions on an early version of the paper. We also thank Hidehiko Masuhara and the members of his research group PRG for discussions on the work.

References

L. Bettini, V. Bono, and B. Venneri. Delegation by object composition. *Sci. Comput. Program.*, 76(11):992–1014, Nov. 2011.

G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP'90*, pages 303–311, 1990.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2.

M. Müller and S. Nishimura. Type Inference for First-Class Messages With Feature Constraints. *International Journal of Foundations of Computer Science*, 11:29–63, 2000.

D. Rémy. Type inference for records in natural extension of ML. In *Theoretical aspects of object-oriented programming: types, semantics, and language design*, pages 67–95. MIT Press, 1994.

N. Schärli, S. Ducasse, O. Nierstrasz, and a. Black. Traits: Composable units of behaviour. *Lecture Notes in Computer Science*, 2743:248–274, 2003.