Anonymous Caching: Managing Duplication and Dependencies

Joel Jones
Department of Computer Science
University of Alabama
jones@cs.ua.edu

1 Intent

Provide an interface for computing objects with derived state only once with the object collecting the cached values having no dependencies on the derived objects.

2 Motivation

Consider a compiler where various analyses may be done on a per-method basis. This analysis information may be needed by code optimizations or in the calculation of other analyses. This information may be expensive to calculate and the user of this information is not concerned with how analyses depend upon each other. If the client is responsible for supplying analysis information to construct the analysis the client desires, then several problems result. The analysis information may be recalculated if there are other users of identical information, leading to unnecessary duplicated work. Also, the dependencies of a particular analysis are exposed, leading to more complex code in the client.

We can solve this problem by having the representation of the method keep track of which analyses have been performed on it and supplying them when requested by either returning them or calculating them. The application of the Anonymous Caching pattern in our compiler example is shown in figure 1. The JavaMethod contains the information needed for analyzing a particular Java method. A DuChain is a collection of information about the definitions and subsequent uses of all variables in a Java method. Each definition (d) is linked with its subsequent uses (u) forming a def-use chain or *du-chain*. Using Anonymous Caching, JavaMethod caches analyses such as DuChains anonymously—there is no specific knowledge of or dependence upon DuChains in JavaMethod.

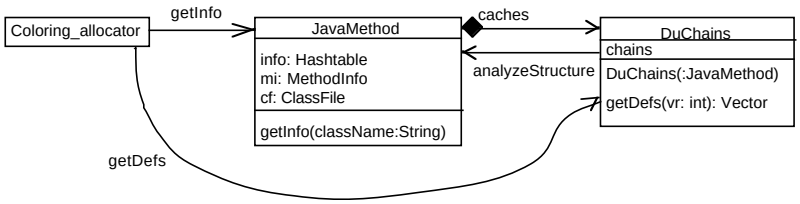

Figure 1: Compiler Example

3 Applicability

Use Anonymous Caching when the following conditions hold:

- There is an object that can have a variable number of objects associated with it.
- The associated objects may be needed by other associated objects or by clients.
- The associated objects can be constrained to be constructed using only the central object as an argument to the associated object's constructor.
- There is at most one instance of a particular class of an associated object for each central object.
- A client needs no information about the dependencies an associated object may have on other associated objects.

Do not use Anonymous Caching if the analysis computation cost is not high enough to outweigh the complexity of using caching.

4 Structure

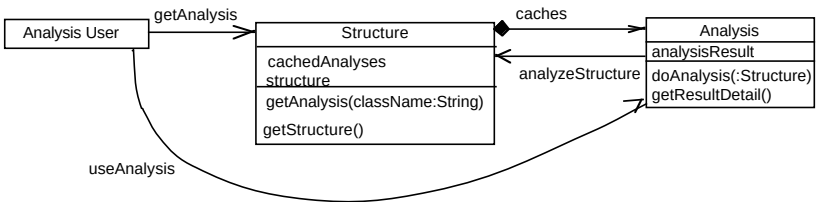

Figure 2: Structure of Anonymous Caching

5 Participants

As shown in Figure 2, there are three kinds of participants in Anonymous Caching.

Structure The object which has information derived from it and holds the cache for the derived data. It provides various accessors methods (`getStructure()`) to access information about it (`structure`.)

Analysis The information derived from Structure. It can be constructed by calling a constructor or method whose only argument is Structure (`doAnalysis()`.) The results of the analysis can be obtained through accessor methods (`getResultDetail()`.)

Analysis User The object which requests Analyses from Structure and subsequently uses the Analyses.

6 Collaborations

In our compiler example, an optimization (Analysis User) may request a specific analysis (Analysis) by its class name. To acquire this analysis, it makes a request of JavaMethod (Structure.) If JavaMethod has had this analysis performed on it before, then the previous value is returned. Otherwise, JavaMethod constructs the analysis object, stores the analysis object in the JavaMethod's cache, and returns the analysis object. The optimization then treats the analysis object as the requested type, with access protocols that may be unrelated to any other analysis object. During the construction of an analysis object, the analysis object will invoke methods on JavaMethod to obtain information that it needs to construct itself. In addition, an analysis object may make requests of the JavaMethod to supply another analysis object during its construction.

7 Consequences

This pattern allows the introduction of new Analyses without having to add code to Structure. By constraining requests for Analyses to identify which class is desired, and by requiring Analyses to have a construction method which takes a Structure as an argument, the Analysis User is simplified. The Analysis User is not required to have knowledge of how to build an Analysis; in particular, what other Analyses the desired Analysis depends upon. The Structure is simplified by keeping track of derived Analyses by keeping an association between the class name and the previously calculated Analysis. By keeping this collection of associations, Structure does not have to implement a new method every time a new type of Analysis is introduced. To introduce a new kind of Analysis, one only need implement the new Analysis, modify any Analysis Users to request it from Structure, and potentially write code to register the construction method. In Section 8.2 we discuss the potential registration code.

If the Structure can change after the analyses are done, then some mechanism must be used to flush appropriate Analyses from the cache. The simplest approach is simply flush all Analyses from the cache. More elaborate mechanisms could be integrated into the interfaces of the Analyses. One approach would be to use Observer, where Structure plays the role of Subject and the Analyses play the role of Observer. [4]

Using Observer, the "changed" message would be parameterized by the type of change in Structure. Then each Analysis would determine if it needs to be recalculated.

Anonymous Caching has the usual negative consequences of using nameless collections of methods, in that the control flow of the program is difficult to follow from static analysis. This applies to both humans and compilers.

8 Implementation

There are two different techniques that may be used in implementing Anonymous Caching. The first technique minimizes the changes that have to be made when a new Analysis is introduced. However, it is restricted to implementation languages that have reflective facilities. The second technique does not require reflection, but does have a greater number of places code has to be inserted to introduce a new Analysis.

Both the reflective and non-reflective implementation techniques of Anonymous Caching share code. Below is the code from the Press Pot compiler, where we first encountered this pattern. [5]

```
public class JavaMethod {
     Hashtable info = new Hashtable();
     ...
     public Object getInfo(String className)  {
         Object theInfo;
         if ((theInfo = info.get(className)) == null) {
             theInfo = createInstanceOf(className);
             info.put(className, theInfo);
         }
         return theInfo;
     }
     private Object createInstanceOf(String className) {
         ...
     }
}

public class DuChains {
     public DuChains(JavaMethod jm) {
         ...
         CFG cfg = (CFG) jm.getInfo("Graph.CFG");
         ...
     }
}
```

The JavaMethod class is acting as the Structure, and both DuChains and Graph.CFG are acting as Analyses. The `getInfo` method of JavaMethod acts as the `getAnalysis` method.

8.1 Reflective Implementation

In languages that allow it, reflection provides the easiest implementation. In such languages, the Structure should have a method that takes the class name of the desired

Analysis. This method is responsible for checking the cache to see if the desired Analysis already exists. If not, then the method uses the reflective facilities of the language to create the requested Analysis and place it into the cache. Once the desired Analysis is obtained, it is returned to the Analysis User by the Structure. Code introduced for an Analysis is in the Analysis itself and in any Analysis Users that use the Analysis.

In the source code for a reflective implementation, such as in Press Pot, the body of `createInstanceOf` would be as follows:

```
private Object createInstanceOf(String className) {
    Class infoClass = Class.forName(className);
    Constructor iConstructor =
        infoClass.getConstructor(jMethodClassArr);
    Object jmArr[] = new Object[1];
    jmArr[0] = this;
    return iConstructor.newInstance(jmArr);
}
```

The Class object for the Analysis is first obtained using `Class.forName()`. The `forName` method is a reflective method that is part of Java's standard set of reflection facilities. Then we find the constructor which takes JavaMethod as an argument, using an array (`jMethodClassArr`) pre-initialized to contain one element, an instance of Class representing JavaMethod. We then construct the argument(s) to the constructor by building a list of the values, represented as an array of Objects, and passing it to `newInstance()`, which creates the requested Analysis. The class `Constructor` is also part of Java's standard reflective facilities.

8.2 Non-reflective Implementation

In languages that do not have reflection, a non-reflective implementation must be used. Of course, it is also possible to implement the techniques described below in a language that does have reflection. A non-reflective implementation of Anonymous Caching is similar to a reflective one, in that there is still the action of creating an instance of a class given the name of the class.

There is a naive approach that immediately comes to mind—have an "if" or "switch" based decision tree implemented in the code of Structure. The leaves of this decision tree would test the requested Analysis name against a literal constant and invoke the corresponding constructor or Factory Method. This does answer the forces of hiding dependencies between Analyses from the Analysis User and of avoiding recalculation of Analyses. However, it doesn't answer the force that Structure not be dependent upon the Analyses.

To lessen the dependence of Structure on Analyses, a method is provided on Structure to register the pair (Analysis name, Analysis Factory). Then, when a request is made of Structure to provide an Analysis, this registry can be searched and the appropriate construction method invoked. No code in Structure needs to be changed when a new Analysis is introduced.

The non-reflective implementation requires more code to implement the cache than does the reflective one. Again, we use Press Pot as an example. First, we need to

declare an interface, AnalysisFactory, which every analysis factory must implement and which has a method taking a JavaMethod which produces an analysis object.

```
public interface AnalysisFactory {
    public Object createAnalysis(JavaMethod jm);
}
```

The next step is to create an implementation of the AnalysisFactory interface for creating DuChains:

```
public class DuChainsFactory implements AnalysisFactory {
    public Object createAnalysis(JavaMethod jm) {
        return new DuChains(jm);
    }
}
```

We also need to extend JavaMethod to contain a global collection of mappings from analysis class names to AnalysisFactories and a method for registering AnalysisFactories with JavaMethod:

```
public class JavaMethod {
    private static Hashtable factories = new Hashtable();
    public static registerFactory(String className,
                                  AnalysisFactory factory) {
        factories.put(className, factory);
    }
}
```

We need a different version of `createInstanceOf` in JavaMethod than we used in the reflective implementation:

```
 private Object createInstanceOf(String className) {
     AnalysisFactory factory = factories.get(className);
     return factory.createAnalysis(this);
 }
```

The next step in implementing Anonymous Caching non-reflectively is to ensure that the registration of AnalysisFactories occurs. The semantics of the implementation language determine whether additional code outside of the Analysis is needed to cause the registration method to be invoked. In languages with static initializers, like C++, the registration method can be invoked by placing a call to the registration process in a static initializer. This initializer can be placed in the source file for the Analysis, thereby centralizing the changes that occur when introducing a new Analysis. The linker takes care of the details of insuring that the static initialization occurs. In languages without such a facility, like Java, the invocation of the registration method has no natural home in the source. Static class initialization in the Analysis does not suffice, as the Analysis's class is not loaded into the VM unless it is "referenced." There are two ways of solving the initialization problem. One is to place an invocation of the registration method in the overall application's startup code. Another is to place an invocation of the registration method in the Analysis User's class initialization code. The second solution is preferred, as it places a dependence on the Analysis in a place where

it already exists, the Analysis User. In either case, the registration method should be idempotent i.e. avoid registering with Structure multiple times.

Below we have an example of the second technique for ensuring registration takes place, with registration code placed in the Analysis User. We have an optimization, Coloring_allocator, which acts as a Analysis User. We can have its class initialization do the job of registration:

```
public class Coloring_allocator {
    static {
        JavaMethod.registerFactory("Graph.DuChains",
                                   new DuChainsFactory());
    }
}
```

The implementation of Hashtable, used by the method `registerFactory` of JavaMethod, ensures that only one AnalysisFactory exists for each Analysis.

If an analysis depends upon another analysis, we also need to insure that the depended upon analysis has its factory registered. In our example, DuChains relies upon CFG, so we must add code to DuChains:

```
static {
    JavaMethod.registerFactory("Graph.CFG",
                               new CFGFactory());
}
```

AnalysisFactories should be implemented as Singletons, but have not in this example, for clarity of exposition; instead we have just allowed the Hashtable of JavaMethod to overwrite any previously registered AnalysisFactory.

9 Known Uses

This pattern is used in Press Pot, a compiler for annotating Java `.class` files with optimization information for use in an annotation-aware Java virtual machine. [5] There, the representation of Java methods, the class JavaMethod, acts as the Structure. The implementation of register allocation, Coloring_optimizer, acts as a Analysis User. In doing the register allocation, Coloring_optimizer has need of definition-use information, represented by the class DuChains, acting as an Analysis. The class DuChains in turn relies upon control-flow information, represented as the class CFG, also an Analysis.

10 Related Patterns

Extensible Attributes is a very similar pattern. [3] Doble, in turn, cites Variable State as being similar to it. [2] Both of these pattern are similar to Anonymous Caching in that they allow the run-time extension of an object's attributes. However, our constraint on the construction method, i.e. requiring a method taking Analysis as an argument, makes the Anonymous Caching pattern easier to use. It allows the Analysis User to

have hidden from it any dependencies the Analysis might have on other Analyses. The resulting code in Analysis User is simpler than it would be without this constraint.

Singleton is a good way of implementing the registry of Analysis Factories in a non-reflective implementation. [4]

Factory Method is used in in the non-reflective implementation to handle the actual object construction. [4] The AnalysisFactory performs the role of Creator and the implementors of Analysis Factory perform the role of ConcreteCreator.

Lazy Initialization is a dual of Anonymous Caching. [1] They share caching as a core element, but the intent and the details are different. In Lazy Initialization, the details of construction are hidden in the state holder from the client, whereas in Anonymous Caching all the details of construction are hidden from the state holder.

11 Acknowledgments

Thanks to Scott Hawker for suggestions and for forcing me to consider the non-reflective implementation techniques.

References

[1] AUER, K. Reusability through self-encapsulation. In *Pattern Languages of Program Design*, J. O. Coplien and D. C. Schmidt, Eds. Addison-Wesley, 1995, ch. 27.

[2] BECK, K. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

[3] DOBLE, J., AND AUER, K. Smalltalk scaffolding patterns. In *Pattern Languages of Program Design 4*, N. Harrison, B. Foote, and H. Rohnert, Eds. Addison-Wesley, 2000, ch. 11, pp. 199–219.

[4] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1994. ISBN 0-201-63361-2.

[5] JONES, J., AND KAMIN, S. Annotating java class files with virtual registers for performance. *Concurrency: Practice and Experience 12*, 6 (2000), 389–406.