Lazy Clone - A Pattern to Improve Performance and Maintainability of Object Cloning

Bruno Cartaxo, Federal Institute of Pernambuco, Brazil
Eduardo Guerra, Free University of Bozen-Bolzano, Italy
Victor Osório, Openet, Brazil
Sérgio Soares, Federal University of Pernambuco, Brazil
Paulo Borba, Federal University of Pernambuco, Brazil

Object cloning is demanded in many situations. Traditional deep cloning is an approach that usually is implemented programmatically, with reflection or serialization. However, those implementations may not fulfill requirements like performance, memory usage, and code maintainability. In this paper, we propose the Lazy Clone Pattern that aims to better reconcile those requirements. We also introduce two Java implementation for the the Lazy Clone, one with Dynamic Proxies (DP) and other based on Aspect-Oriented Programming (AOP).

Categories and Subject Descriptors: D.1.5 [**Programming Technique**] Object-oriented Programming

Additional Key Words and Phrases: Lazy Clone, Deep Clone, Object Clone, Performance, Maintainability

1. CONTEXT

Cloning objects with object-oriented languages is error-prone and hard to maintain activity and can also hinder system performance. Some inherent flaws have been broadly discussed, especially in the Java platform [Bloch 2008]. Security issues are also presented by Jensen et al. [Jensen et al. 2011]. However, clone objects cannot be avoided in situations such as in the prototype design pattern [Gamma et al. 1995], with the Copy-On-Write (COW) strategy, some graph transformations [Drewes et al. 2006], and others.

Two well-known approaches to clone objects are the Shallow and Deep clones [Jensen et al. 2011]. The first is typically a naive strategy since it clones an object without its dependencies, which might cause hard-to-detect side effects during runtime. On the other hand, the deep approach clones the entire dependency graph of an object, promoting independence between the clone and its original object. Nevertheless, that approach can be CPU and memory-intensive, especially when parts of the dependency graph are unnecessary.

A better approach would provide independence between the clone and its original, but also avoid cloning the whole object graph upfront [Cartaxo et al. 2015]. To better understand the problems inherent to the Shallow and Deep Cloning approaches, the following subsections detail their pros and cons.

1.1 Shallow Clone

This approach creates an object of the same type as the original, then copies each field bit-by-bit. That is simple and straightforward but has some undesirable drawbacks. If the copied fields are references to other objects, the bit-by-bit copy makes the clone point to the same objects pointed by the original. To illustrate that situation, suppose we have an object A and want to shallow copy it. First, we create an object B and copy all A's fields to B. Once reference fields store memory addresses, B's reference fields will store the same addresses of A's fields instead of their values. Consequently, B will point to the same objects that A points to, as depicted in Figure 1. Therefore, modifications on A will have side effects on B, and vice versa [Jensen et al. 2011]. In languages like Java, a shallow clone can be easily implemented with the clone method existing in all objects, but not without the extensively documented flaws related to this approach[Bloch 2008].

1.2 Deep Clone

This approach aims to overcome the side effects on reference fields inherent to shallow cloning. It clones all reference fields recursively until the entire field graph is traversed, instead of copying their memory addresses like the shallow approach [Jensen et al. 2011]. For instance, suppose that object B is a deep clone of A. The reference fields of B do not point to the same objects that A points to, as shown in Figure 2. Nevertheless, the deep clone approach can introduce performance penalties and high memory usage by solving the side effects problem.

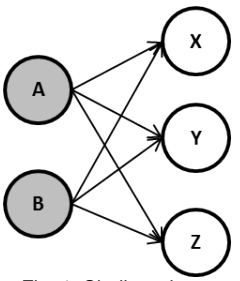

Fig. 1: Shallow clone.

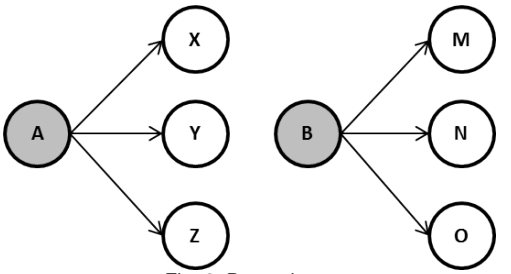

Fig. 2: Deep clone.

2. MOTIVATING EXAMPLE

Consider a system that retrieves from the database an object structured as a complex graph of dependencies. This object should be presented in a graphical interface to the user so that s/he can browse and edit its information that would be saved later. The developers decided that using a clone of the original object is a suitable solution because it would be edited, and it is not desirable to change the original object. The first solution was to create a deep clone of the object, but this operation took a significant time due to the dependencies' complexity and size. Moreover, only a tiny part of the object was accessed and modified by the user.

3. PROBLEM

How to balance maintainability and performance when deep cloning an object?

4. SOLUTION

Create a clone from the root object in the structure and postpone the clone of objects that compose it to the moment that they are first accessed.

An alternative to the presented approaches is lazy cloning, a middle ground between shallow and deep ones. Each reference field is cloned on demand when they are required. Lazy clones consume only the necessary memory as a consequence of cloning just needed fields. The deep approach, on the other side, clones the entire graph of objects even when many are not used. Additionally, the lazy approach is potentially faster than the deep counterpart once it usually creates fewer objects.

The main idea behind the lazy cloning approach is to create an object with all reference fields set to null and the primitive and immutable fields initialized with the same values as the original object. Reference fields pointing to null indicate that the fields were not cloned. Consequently, when a null field is accessed, it is cloned unless the corresponding original field is also null. Moreover, each clone must maintain a reference to its original object to access the original's fields values when demanded.

Figure 3 presents a sequence diagram of the Lazy Clone Pattern independent from any specific implementation. The lazy clone interceptor represented in the diagram works as a proxy that intermediates the access of the lazy clone to the client code. In other words, the access to the interceptor is transparent, and for the client code, it seems that it is invoking methods directly on the cloned object.

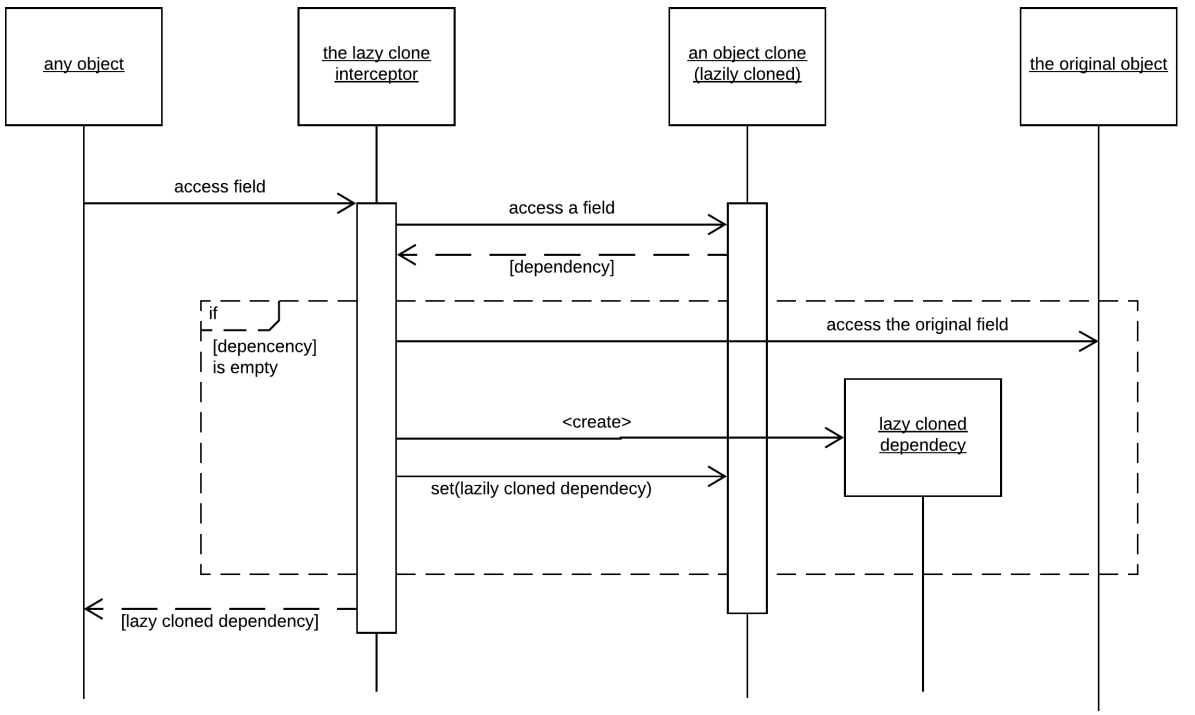

Fig. 3: The solution sequence diagram.

It is important to note one consistency characteristic about lazy cloning. Since fields are cloned when demanded, a lazy clone shares the fields graph with the original object until the fields are accessed. Thus, if one modifies a field of the original object that was not cloned yet, the lazy clone will clone this new value in the future, not the value once held on the cloning moment. This characteristic is not a problem for the prototype design pattern that creates unchangeable prototypical instances but should be considered if it is not desirable behavior.

5. CONSEQUENCES

—(+) The cloned object can be returned fast since part of the cloning will happen on demand.

—(+) Parts of the object that were not accessed will not be cloned, improving the performance.

—(-) The overall performance might be reduced when all the fields of the cloned object are needed.

—(-) If the original object change, since parts of the clone are copied on demand, the clone created before the change might have the new value. [1]

[1] A possible work around for this problem (out of the scope of this pattern) is using annotations like mechanism to mark fields that need to be eagerly cloned. However, if the number of field marked as eager is too high, you may lose the benefit of lazy cloning.

—(-) A more general and reusable implementation of the pattern relies on reflective programming, which is complex to create.

6. IMPLEMENTATION STRATEGIES

Before we introduce possible implementations for the Lazy Clone Pattern, we present the most common implementations of deep clone in order to highlight their pros and cons. So, we can contrast deep with the lazy cloning implementations we are proposing.

6.1 Deep Clone Implementations

6.1.1 *Programmatically (P).* This implementation defines one method per class intended to be cloned. That method creates a new object of the same type of the original dynamically allocating it. After, copies all primitive and immutable fields from the original object. Finally, it calls the clone method for each of the reference fields. This creates a dedicated object graph for the clone, separated from the original object. It can be implemented with a method or a constructor per class.

This implementation does not require any framework or library; only the native constructs of the programming language. On the other hand, its implementation is not only tedious and error prone, but also becomes progressively hard to maintain when the number of classes, hierarchy depth and number of fields grow. For instance, if a new field is added to a class, or any of its parents, the clone method needs to be updated. Otherwise, it can lead to hard to find errors. This implementation has also a restriction that impedes it to clone objects from third-party libraries. This occurs since the developer does not have control of the source code and consequently cannot implement the cloning methods.

Pros of implementing Deep Clone Programmatically:

—It does not demand third-party libraries to be implemented.

—It usually has a good performance since it does not introduce any extra procedure apart from cloning each object field.

Cons of implementing Deep Clone Programmatically:

—For cloning objects, one may have to implement various methods and keep always maintaining them when adding or removing class attributes, depending on the implementation.

—Objects of classes from third-party libraries cannot be cloned, depending on the implementation.

—It may consume more memory than necessary, as any deep clone implementation.

6.1.2 *Reflection (R).* Defines a generic method that deals with objects from any class. It inspects field-by-field the original object with reflection cloning each of them. This can be achieved implementing a method from scratch, which can be hard to cover all special cases, like immutable objects, enums, and others. Or implemented through a library like Java Deep Clone Library[2] that treats all cases.

This implementation cuts off the tedious job to create one method per class, like in the dynamic allocation. Consequently, improves code maintainability centralizing the clone logic in just one method. Another advantage is that with reflection it is possible to clone any object, even from third-party classes. Nonetheless, reflection is well known as an expensive technology possibly making its cost prohibitive for classes with complex structures.

Pros of implementing Deep Clone with Reflection:

[2] https://code.google.com/p/cloning/

—Improves code maintainability by centralizing the clone logic in just one method.

—It is possible to clone any object, even from third-party classes.

Cons of implementing Deep Clone with Reflection:

—Reflection is well known as an expensive technology, possibly making its cost prohibitive for cloning objects with complex structures.

—It depends on third-party libraries.

—It may consume more memory than necessary, as any deep clone implementation.

6.1.3 *Serialization (S).* This implementation relies on a serialization mechanism that translates objects into a format that can be posteriorly reconstructed. It has only one generic and centralized method to clone any classes of objects. It serializes the original object and put the bytes in a memory buffer. Afterwards, it reads the bytes from the buffer, deserializes it, creating a deep clone of the given object. This solution can be easily implemented from scratch. For Java platform, the Apache Commons Lang[3] library provides a class that do it.

Like with reflection, the serialization implementation is easy to maintain due to its generic treatment to clone any type of object. However, this solution has some important drawbacks. Serialization is a very expensive procedure, thus it can introduce unacceptable performance penalties for critical code. In Java platform, this implementation has also some limitations related to third-party classes. It can deal only with classes that implement the `Serializable` Java interface. Thus, any object from a class outside the developer's control that do not implement the mentioned interface, cannot be cloned.

Cons of implementing Deep Clone with Serialization:

—Improves code maintainability by centralizing the clone logic in just one method.

Cons of implementing Deep Clone with Serialization:

—Serialization is well known as an expensive technology, possibly making its cost prohibitive for cloning objects with complex structures.

—Objects of classes from third-party libraries cannot be cloned.

—It depends on third-party libraries.

—It may consume more memory than necessary, as any deep clone implementation.

6.2 Lazy Clone Implementations

Here we present two implementations of the Lazy Clone Pattern based on the concept of Dynamic Proxy (DP) and Aspect Oriented Programming (AOP). Following we detail the implementations.

6.3 Dynamic Proxy

The lazy cloning approach defers the process of cloning objects fields to the moment they are needed. A way to implement this behavior could be introduce verification in all get methods of all classes in order to check if the required field was already cloned. This is clearly not practical and hard to maintain.

[3] http://commons.apache.org/proper/commons-lang/

Thus, we decided to employ the dynamic proxy technology to implement our first version of lazy approach. We use Byte Buddy[4] library, since the default Java dynamic proxy framework does not support creation of proxies based on classes, only interfaces[5]. Behind the scenes, Byte Buddy dynamically creates classes extending the classes we want, and implementing methods defining the desired behavior at bytecode level. In our case, we will create proxies objects that act like the originals, but with additional behavior.

Listing 1 shows the class that clones objects through the dynamic proxies implementation. Error handling and auxiliary methods are omitted due to space restrictions. The clone method (line 45) is the entry point to the cloninig process. It creates a new wrapper class with the GetterLazyLoadInvocationHandler (Listing 3 and SetterLazyLoadInvocationHander setted according to its method definition. To avoid reflection calls, we define the object as an instance of ProxyableObject (Listing 2) where the already cloned fields and source objects are also accessed by methods. As we can see, when subclass we need to define all methods that we want to intercept (lines 22-35), all new fields we want to create (lines 30,31), then we need to load the new class on the same classloader from the original class.

Listing 1: Lazy cloning implementation with dynamic proxies.

```
1  public class LazyObjectProxyCloner<T> {
2    private static final String FIELD_LAZY_CLONED_FIELDS = "$$_lazy_cloned_fields";
3    private static final String FIELD_LAZY_SOURCE = "$$_lazy_source";
4    private Constructor<? extends T> proxyConstructor;
5
6    public LazyObjectProxyCloner(Class<T> objectClass) {
7      try {
8        Set<Field> copyableFields = new HashSet<>();
9        Set<Field> clonableFields = new HashSet<>();
10       ClassInspector.getAllFields(objectClass).forEach(field -> {
11         if (ClassInspector.isPrimitive(field.getType()) ||
12             ClassInspector.isImmutable(field.getType())) {
13           copyableFields.add(field);
14         } else {
15           clonableFields.add(field);
16         }
17       });
18       Class<? extends T> proxyClass =
19             new ByteBuddy().subclass(objectClass).implement(ProxyableObject.class)
20                       .defineConstructor(Visibility.PUBLIC).withParameter(objectClass)
21                       .intercept(createConstructor(objectClass, copyableFields)
22                       .method(ElementMatchers.anyOf(clonableFields.stream()
23                                       .map(ClassInspector::getGetter)
24                                       .toArray(Method[]::new)))
25                       .intercept(InvocationHandlerAdapter.of(new
       GetterLazyLoadInvocationHandler()))
26                       .method(ElementMatchers.anyOf(clonableFields.stream()
27                                       .map(ClassInspector::getSetter)
28                                       .toArray(Method[]::new)))
29                       .intercept(InvocationHandlerAdapter.of(new
       SetterLazyLoadInvocationHandler()))
30                       .defineField(FIELD_LAZY_SOURCE, objectClass, PRIVATE)
31                       .defineField(FIELD_LAZY_CLONED_FIELDS, Set.class, PRIVATE)
32                       .method(ElementMatchers.named("$$getLazySource"))
```

[4]https://bytebuddy.net/

[5]https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html

```
33                        .intercept(FieldAccessor.ofField(FIELD_LAZY_SOURCE))
34                        .method(ElementMatchers.named("$$getClonedFields"))
35                        .intercept(FieldAccessor.ofField(FIELD_LAZY_CLONED_FIELDS))
36                        .make().load(objectClass.getClassLoader(), ClassLoadingStrategy.
      Default.INJECTION)
37                        .getLoaded();
38
39          proxyConstructor = proxyClass.getConstructor(objectClass);
40        } catch (NoSuchMethodException e) {
41          throw new RuntimeException("Cannot␣create␣proxy␣class!", e);
42        }
43      }
44
45      public T clone(T obj) {
46        try {
47          return proxyConstructor.newInstance(obj);
48        } catch (Exception e) {
49          throw new RuntimeException(e);
50        }
51      }
52    }
```

Creating a Dynamic Proxy is not straightforward. The class should be dynamic defined, which means that every element from a class is defined calling Byte Buddy API. On Listing 1, it starts by subclassing the original class and adding a helper interface to avoid reflection calls. Fields, methods and constructors should be defined using a Builder and then, for methods and constructors, its behaviour should be defined using interceptors. To define a Method, first, it needs to select it using ElementMatchers and then define its related interceptors, as it defines the getters behaviour on lines 22-29.

On wrapper construct, we copy all immutable and primitive fields. store the original object in a source field and create a set of non copied fields. The dynamic proxy will call the original getters and setters for these copied values and the other one will be proxied to its respective InvocationHandler. Each invocation handler should check if the fields have already been cloned and clone then if necessary.

At the end (lines 30-35) two methods are defined to avoid reflection calls improving the general performance. The first method is $$getLazySource will return the object being cloning. The second one is $$getClonedFields that will provide a set of fields that were not cloned yet. These methods will be used on interceptors GetterLazy-LoadInvocationHandler and SetterLazyLoadInvocationHandler.

Listing 2: Proxyable object interface, it should be imnplemented by Dynamic Proxies.

```
1  public interface ProxyableObject {
2      public Set<Field> $$getClonedFields();
3      public Object $$getLazySource();
4  }
```

ProxyableObject, on Listing 2, is an interface that all Dynamic Proxy should implement. Its purpose is to avoid reflection calls on interceptors improving the general performance from the dynamic proxy. The interceptors are defined on 3 and 4. The intercept behaviour should be simple to avoid performance penalties. In both cases, the field related to the method is found by a helper class, if this field is not on cloned set, this means that it should be cloned.

Listing 3: Invocation handler for getters

```
1  public class GetterLazyLoadInvocationHandler implements InvocationHandler {
2
3    @Override
4    public Object invoke(Object object, Method method, Object[] args) throws Throwable {
5      ProxyableObject proxy = (ProxyableObject) object;
6      Field field = ClassInspector.getFieldByGetter(method);
7      Set<Field> clonedFields = proxy.$$getClonedFields();
8      if (!clonedFields.contains(field)) {
9        field.setAccessible(true);
10       field.set(proxy, LazyCloneFactory.this.clone(field.get(proxy.$$getLazySource())));
11       clonedFields.add(field);
12     } else {
13       field.setAccessible(true);
14     }
15     return field.get(proxy);
16   }
17 }
```

Listing 4: Invocation handler for setters

```
1  public class SetterLazyLoadInvocationHandler implements InvocationHandler {
2
3    @Override
4    public Object invoke(Object object, Method method, Object[] args) throws Throwable {
5      ProxyableObject proxy = (ProxyableObject) object;
6      Field field = ClassInspector.getFieldBySetter(method);
7      Set<Field> clonedFields = proxy.$$getClonedFields();
8      if (!clonedFields.contains(field)) {
9        field.setAccessible(true);
10       field.set(proxy, LazyCloneFactory.this.clone(field.get(proxy.$$getLazySource())));
11       clonedFields.add(field);
12     } else {
13       field.setAccessible(true);
14     }
15     field.set(proxy, args[0]);
16     return null;
17   }
18 }
```

This implementation has all the benefits of a lazy cloning approach and also can clone third-party objects, except the ones from a class that cannot be subclassed. This occurs because Byte Buddy create proxies subclassing the original class.

On the other hand, this implementation can provoke performance penalties, since it manipulates bytecodes at runtime, but this penalty can be avoided by caching the LazyObjectProxyCloner instance creating it at startup time. It also imposes that objects intended to be cloned have getters and setters methods for each field. And also that to access those fields one must use the getters and setters methods, no direct field access would be allowed. Additionally, the class must have have a constructor with no parameter.

Another drawback is that all direct accesses to private fields inside classes must be changed by the corresponding get methods. This guarantees that any access to one of the clone fields will be handled by the dynamic proxy interceptor, avoiding inconsistent states and null pointer exceptions. Direct accesses to protected fields in all subclasses must also be changed for the same reason.

The tedious job to exchange fields accesses by get methods can be alleviated by automatic refactoring techniques. However it demands the development and maintenance of an automatic routine, as well as integration of it to the system build in order to deal with evolution of the system source code.

Cons of implementing Lazy Clone with Dynamic Proxies:

—This implementation has all the benefits of a lazy cloning approach and also can clone third-party objects.

Cons of implementing Lazy Clone with Dynamic Proxies:

—this implementation can provoke performance penalties, since it manipulate bytecode at runtime.
—It also imposes that objects intended to be cloned have getters and setters methods for each field. It also imposes that any access to any object field is made through those getters and setters methods.
—It depends on third-party libraries.
—Modifications in the original object may produce undesirable side effects depending on the situations, as any lazy clone implementation.

6.4 Aspect-based

We decided to pursue an alternative technology in order to deal with the performance penalties emerged from the dynamic proxies implementation, as well as the impacts for exchange all fields accesses to get method calls.

Thus, we adopted aspect-oriented programing (AOP) with AspectJ[6], which meet those two requirements. The performance improvement came from the fact that AspectJ can anticipate bytecode manipulation to compile-time, removing overhead from the runtime as occurs with dynamic proxies. The second requirement is met through AspectJ join points that intercept fields accesses. This eliminates the effort to exchange all fields accesses by get method calls.

Listing 5 shows the aspect that intercepts all objects intended to be cloned. Error handling and auxiliary methods are omitted due to space limitations. The `fieldAccess` pointcut (line 3) intercepts all accesses to instance fields of any object that directly or indirectly implements the `LazyCloneable` interface. This is a tag interface that any object intended to be lazy cloned by the AOP implementation must implement. Line 4 is an inter-type declaration that introduces a field to all `LazyCloneable` classes in order to hold a reference to the original object. Another inter-type declaration, in line 6, defines that all `LazyCloneable` objects have a `cloneOf` method. This method is the entry point for the clone process. It takes the given original object, set in the original field, and copies each primitive and immutable fields. Thus, to clone an object through the AOP implementation, the developer only needs to instantiate an empty object, call the `cloneOf` method passing the original object, and then the new object is now a lazy clone of the original.

The advice (line 16) handles all fields accesses, as defined by the aforementioned pointcut. It first checks whether the intercepted object is a clone or not. Proceeds to the natural flow in the latter case to avoid performance penalties for non-clone objects (line 17). If the object is a clone, then it verifies if the requested field is primitive, immutable or was already cloned, returning the field value without any major processing (line 25). Otherwise, the accessed field was not yet cloned and will be cloned according to its type (lines 27-37).

Listing 5: Lazy cloning implementation with Aspect-Oriented Development.

```
1  public aspect LazyClone {
2
```

[6] http://www.eclipse.org/aspectj/

```
3  public pointcut fieldAccess() : get(!final !static * LazyCloneable+.*) && !within(
       LazyClone);
4  private Object LazyCloneable.original = null;
5
6  public void LazyCloneable.cloneOf(Object original) {
7    if (this.getClass().equals(original.getClass())) {
8      this.original = original;
9      for (Class<?> clazz = original.getClass(); clazz != null; clazz = clazz.
       getSuperclass()) {
10       for (Field field : clazz.getDeclaredFields()) {
11         if (isPrimitive(field) || isImmutable(field)) {
12           field.setAccessible(true);
13           field.set(this, field.get(original));
14 }}}}}
15
16 Object around(Object target) : target(target) && fieldAccess() {
17   if (((LazyCloneable)target).getOriginal() == null) {return proceed(target);}
18   else {
19     LazyCloneable clone = (LazyCloneable)target;
20     String fieldName = thisJoinPointStaticPart.getSignature().getName();
21     Field field = getDeclaredField(clone, fieldName);
22     Class<?> fieldClass = field.getType();
23     field.setAccessible(true);
24     if (isPrimitive(field) || isImmutable(field) || field.get(clone) != null) {
25       return proceed(clone);
26     } else {
27       Object fieldOriginalValue = field.get(clone.getOriginal());
28       if (fieldOriginalValue == null) {
29         return null;
30       } else if (LazyCloneable.class.isAssignableFrom(fieldClass)) {
31         LazyCloneable fieldClone = cloneCloneable((LazyCloneable) fieldOriginalValue);
32         field.set(clone, fieldClone);
33         return fieldClone;
34       } else if (Collection.class.isAssignableFrom(fieldClass)) {
35         Collection fieldClone = cloneCollection((Collection) fieldOriginalValue);
36         field.set(clone, fieldClone);
37         return fieldClone;
38 }}}}}
```

This implementation tends to improve code maintainability compared to the dynamic proxies, once it does not need to exchange fields accesses by get method calls. It also, avoids performance penalties by manipulate bytecodes at compile-time instead of runtime. On the other hand, this implementation cannot clone third-party objects once their classes are supposed to implement the `LazyCloneable` tag interface, and also be weaved by the lazy clone aspect.

Cons of implementing Lazy Clone with AOP:

—The performance improvement came from the fact that AspectJ can anticipate bytecode manipulation to compile-time, removing overhead from the runtime as occurs with dynamic proxies.

—The second requirement is met through AspectJ join points that intercept fields accesses. This eliminates the effort to exchange all fields accesses by get method calls as occurs with dynamic proxies.

Cons of implementing Lazy Clone with AOP:

—Objects of classes from third-party libraries cannot be cloned.

—Modifications in the original object may produce undesirable side effects depending on the situations, as any lazy clone implementation.

7. RELATED PATTERNS

The following patterns are related to the Lazy Clone Pattern: Copy On Write Strategy (COW), Prototype.

The pattern `Proxy` can be used to represent the fields that were not copied yet. When the value is first accessed, the `Proxy` access the original object and copy the value of that reference.

REFERENCES

Joshua Bloch. 2008. *Effective Java (2Nd Edition) (The Java Series)* (2 ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.

Bruno Cartaxo, Paulo Borba, Sergio Soares, and Helio Fugimoto. 2015. Improving Performance and Maintainability of Object Cloning with Lazy Clones: An Empirical Evaluation. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–8. DOI:http://dx.doi.org/10.1109/ESEM.2015.7321202

Frank Drewes, Berthold Hoffmann, Dirk Janssens, Mark Minas, and Niels Van Eetvelde. 2006. Adaptive Star Grammars. In *Graph Transformations*. Lecture Notes in Computer Science, Vol. 4178. Springer Berlin Heidelberg.

Erich Gamma, Richard Helm, and others. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Thomas P. Jensen, Florent Kirchner, and David Pichardie. 2011. Secure the Clones - Static Enforcement of Policies for Secure Object Copying.. In *ESOP (Lecture Notes in Computer Science)*, Vol. 6602. Springer, 317–337.