A Static Alternative To Java Dynamic Proxies

Abstract
The Proxy design pattern is often used in applications, but induces generally an implementation overhead. To simplify the developer work, the Java Dynamic Proxies incorporate this pattern as a feature of the Java language.

The Java Dynamic Proxies provide an answer based on reflection to automate operations on the model. The dynamic proxies are used to create a proxy at run-time that implements a set of interfaces. We show that dynamic proxies suffer from lack of static reliability, complex usage requiring introspection and performance overhead.

We propose a static approach introducing the proxy class in the model at compile time. Instead of using services of java.lang.reflect, we rely on the services of the compiler to generate a type-safe proxy customizable by specialization.

This static approach highlights the limitations of the Java type system. We propose some solutions to this limitation that are applicable to the case of static proxy and without extending of the type system of the Java language.

To verify usability of this solution, we implement static proxies in the Java OpenJDK 7 compiler and apply it to the Java Collection library. Static proxies allow to suppress trivial code like naive forwarding and can reduce methods to be written by the developer by 35%.

Categories and Subject Descriptors D.3.2.p [Programming Languages]: Object-oriented languages; D.3.3 [Programming Languages]: Language Constructs and Features; D.3.3.n [Programming Languages]: Patterns

General Terms dynamic proxies, static proxies

Keywords proxy generation, dynamic proxies, static proxies, language construct, specialization, Java.

1. Introduction
The Proxy pattern is a good practice that is frequently used by designers and developers. Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object rather than a simple pointer.

Manual implementation of proxy is often tedious, some have therefore proposed to include it as a programming language feature [3, 7]. Furthermore, according to Soukup [12] and Bosch [1], a pattern used in the design should be implemented as an entity easily noticeable in the programming language.

The Java Dynamic Proxies [5, 13] (JDP) provide an answer based on reflection to automate operations on the model. The dynamic proxies are used to create a proxy at run-time that implements a set of interfaces subject. The limitation of this approach is that using reflection induces a performance overhead, a higher complexity and less static reliability.

We propose a static approach introducing the proxy class in the model at compile time. Instead of using services of java.lang.reflect, we rely on the services of the compiler to generate a type-safe proxy customizable by specialization.

Unlike dynamic approaches, the static approach suffers from limitations of the static typing. Particularly to type references which must represent a union type as a set of subject interfaces used to define the proxy.

The question is: How to type the reference to the proxy when the proxy class will be added at compile time? In this paper, we propose some alternatives to handle the lack of expressiveness of static typing with automated proxies in Java. Static proxies do not require an extension of the type system of the Java language.

The rest of this paper is organized as follow. Section 2 presents the Java Dynamic Proxies and section 3 discusses the benefits and disadvantages of dynamic approach. Sections 4 and 5 draws an alternative based on a static proxy. Typing problems of proxy references are discussed in these sections. Section 6 lists details of the static proxies implementation in the OpenJDK 7 javac compiler. Section 7 gives a concrete example of use of the static proxies for wrappers of the collection Java library. Finally, section 8 presents the previous work on proxies generation and section 9 gives a conclusion of this approach.

2. Java Dynamic Proxies
Java Dynamic Proxies (JDP) were introduced in version 1.3. Dynamic proxies use reflection to proxify existing classes at run-time.

2.1 Presentation
As in the Proxy design pattern, a dynamic proxy class is a class that implements a subject interface list. The (dynamic) proxy class can be used to create a proxy object that delegates its behavior to other objects.

The proxy class is created at run-time through the API java.lang.reflect and requires no pre-generation at compile time.

Conforming to Design Patterns [6], in the rest of this article, the term proxy refers to the class that redirect messages to an encapsulated object. The real subject is the object that provides the implementation to use. Subjects are the interfaces implemented by both the proxy and the real subject.

2.2 Dynamic Proxy Class and Instance
Proxy classes are public, final, and not abstract classes that extend java.lang.reflect.Proxy and implement a list of subject interfaces.

The Proxy.getProxyClass method is used to obtain the proxy class given a class loader and a list of interfaces to be implemented by the proxy. If a proxy class already exists for the pair class
loader / interfaces list (in the specified order), the existing proxy class is returned. Otherwise, a new class is created and loaded in the specified class loader.

Every interface types given must be visible through the specified class loader. To avoid invocation conflicts, two subject interfaces cannot have a method with the same name and parameter signature with a different return types.

A proxy instance is an instance of a proxy class that can be obtained by calling the Proxy.newProxyInstance method or the public constructor of the proxy class.

Each proxy instance has an associated invocation handler object that redirect interface method invocations to the real subject.

### 2.3 Invocation Handlers

An invocation handler is an implementation of the interface java.lang.reflect.InvocationHandler. Method invocations on an instance of a dynamic proxy class are dispatched to the single method InvocationHandler.invoke in the instance invocation handler. Methods are represented by a java.lang.reflect.Method object identifying the invoked method and an array of type Object containing the argument values passed. The invocation handler is then responsible for redirecting the call to the real subject.

The value returned by the invoke method will become the return value of the method invocation on the proxy instance. If an exception is thrown by the invoke method, it will be also thrown by the method invocation on the proxy instance.

Some methods defined in the class java.lang.Object must be processed by the actual implementation and not the proxy instance. An invocation of the hashCode, equals, or toString methods on a proxy instance will be delegated to the invocation handler in the same manner as interface method invocations. Other public methods of a proxy instance inherited from Object are final, thus they are not overridden by the proxy class.

### 2.4 Concrete Use

Figure 1 shows a small hierarchy example used in a collection library. Class Collection provides read access methods through iterator method introduced in Iterable interface. Collection also provides write access methods add and remove.

We need to offer an implementation of the Iterable interface, but for time and space considerations, we don’t want to recopy all values contained in the Collection. We create a proxy based on Iterable which maintains a reference to the real collection Collection and forwards message iterator to it. Thus, all access to other methods rather than those defined in Iterable are statistically denied by the compiler. Figure 2 gives the implementation of the Dynamic Proxy used to proxify Collection. Note that the example is for a simple direct proxy where the method is forwarded as this to the real subject with the original arguments. However, a more sophisticated proxy can perform more complex things by inspecting realSubject, m, and args.

```java
interface Iterable {
    public Iterator iterator();
}

class Collection implements Iterable {
    public Iterator iterator() {
        ...
    }
    public void add(Object e) {
        ...
    }
    public void remove(Object e) {
        ...
    }
}
```

---

**Figure 1.** Hierarchy example based on a collection library.

**Figure 2.** Example of Java Dynamic Proxy applied on the base hierarchy.

### 3. Assessment

Bosch [1] identifies four general issues in the implementation of design patterns: implementation overhead, reusability, code traceability, and the self problem. However we generalize the code traceability to overall traceability (code, run-time and documentation) and the self-problem to the issue of trade-offs facilities. We also add three other issues: static reliability, code evolution, and performance.

In the rest of this section, we examine the impact of these issues with Java Dynamic Proxy versus a standard manual static implementation of the Proxy pattern.

#### 3.1 Implementation Overhead

The quantity and the complexity of the code that need to be written to implement the Proxy pattern should be minimal.

In an ideal proxy mechanism, the trivial part of the proxy should not require some specific implementation.

Manual implementation is not ideal since proxies need the writing of many methods with trivial behavior to redirect calls.

JDP remove the burden of writing each method that just redirects the call. However, the complications are all concentrated in the implementations of the InvocationHandler interface that may require complex code even in simple cases.

#### 3.2 Reusability

With manual implementation, each proxy class has to be re-implemented for each situation and is left to the responsibility of the developer.
With JDP, most of the proxy internals are done through the same `java.lang.reflect.Proxy` class that is reused for each specific proxy.

The implementations of the `InvocationHandler` interface can also be reused as it for a different proxy class. The implementation can also belong to a complex class hierarchy in order to factorize more code.

### 3.3 Traceability

With manual implementation, the pattern is not noticeable among all other classes in the source code. This has a direct effect on application readability, understanding and maintenance. The same observation can be done about the code documentation traceability.

With JDP, proxies are instances of `java.lang.reflect.Proxy` thus it is easy at run-time to spot which object is a proxy. However the code traceability is lost in the specific implementations of the `InvocationHandler` interface since there is no static link between an implementation and the subject. Especially, the method dispatch is done based on a `java.lang.reflect.Method` object.

### 3.4 Trade-offs Facilities

The generated proxy implementation should be adaptable to custom contexts and specific needs. Moreover, compromises shall not result in a too important (or too complex) amount of code to be written by the developer.

The *Self problem* [1] is an example of trade-off facilities. It occurs when the real subject implementation internally uses a reference to `self`. But in the case of proxification, we would like it replaced its references by `Proxy`.

Both the manual and the JDP approach of proxies can deal with trade-offs facilities since the programmer can freely decide what is the behavior of each method of the proxy. Although, with JDP, it requires more complex code that deals with introspection.

### 3.5 Static reliability

With manual implementation, the proxy class specializes the subject interface, thus all methods that need to be implemented/delegated is statically known (and an incomplete proxy will be refused by the compiler). However, instantiation of the proxy class is safe and straightforward.

With JDP, all this static information is lost. Every bad call of `Proxy.newProxyInstance` leads to a run-time error while in most cases, the types could have been verified statically. The exceptions related to bad method invocations occur at run-time rather than compile time which complicates the application debugging. The type system is weakened by the need to dynamic cast on generated proxy objects.

### 3.6 Code Evolution

What append if the subject or the real subject evolve?

With an ideal solution, the proxy will evolve automatically if possible, and when it is not possible, alert the programmer that the proxy is not synchronized (and that some manual intervention is required).

With manual implementation, the evolution is completely manual. However, the static typing will help to catch obvious cases of desynchronization since the manual proxy class will be refused by the compiler if it is not conform with the evolved subject interface or if it incorrectly used the evolved real subject.

With JDP, evolution may have various impact according how the `InvocationHandler` is implemented. In the best case, evolution will be automatically and correctly handled; for instance new methods in the subject interface will be correctly forwarded to the real subject. In the worth case, the program will crash at run-time because of a bad dynamic cast or a failed dynamic verification.

### 3.7 Performance

The use of reflection and dynamic verifications in JDP implies *de facto* a performance overhead. Mainly due to the invocation handler.

### 3.8 Conclusion on the Java Dynamic Proxies

The Java Dynamic proxies provide an overall satisfactory answer to the proxy pattern generation issues, but suffer from some limitations.

First the lack of static guarantees since all the static information is lost and every bad call leads to a run-time error.

Secondly the complex usage requiring introspection, specially in the implementations of the `InvocationHandler` interface.

Thirdly the performance overhead through the use of reflection.

## 4. Static Proxy Classes

We propose a static alternative to the generation of proxies. We keep solutions introduced by JDP adapting them to the static world. Our proposal is consistent with the dynamic proxies and the rules of the Java language.

Our approach have to handles the problems of the proxy pattern and those of JDP identified in the previous section. Furthermore, we want to compensate the loss of the flexibility of run-time generation by a safer typing, better performances and more expressiveness. The solution must also limit the amount of trivial code to be written by the developer.

Finally, to qualify as an acceptable alternative, the static solution must be compatible with the existing dynamic proxies specification.

### 4.1 Static Approach

Rather than using reflection to generate the proxy class, we rely on the services of the compiler. The proxy is introduced into the model of the application at compile time. The use of the class loader or the invocation handler is no longer necessary. The type constraints checking is delegated to the compiler and does not require dynamic tests or casts.

One key point of our approach is that even if we extend the Java language (and modify the Java compiler), we do not modify the virtual machine.

The model introduction offers some advantages:

- We can benefit from the compiler work and its type checking as code generation;
- We keep the original source code clean of trivial code reducing total amount of lines;
- We keep the possibility to change and regenerate the proxy implementation at each compilation;
- We allow the software engineer to inherit and customize the introduced proxy class;

A new Proxy class is inserted in the model at the request of the developer for each set of subject interfaces specified. A single class is created for each set of interfaces.

The Proxy class contains a reference to the real subject instance to which delegate its implementation. It provides a naive implementation for all methods defined in the subjects it implements.

This proxy can be used by the developer as this in obtaining a new instance. Or specialized to redefine the default behavior (see section 5.)
4.2 Syntax
The figure 3 presents the syntax of a static proxy instantiation. The `proxy` keyword is used to indicate to the compiler that the following type reference must be redirected to the generated proxy class rather than the subject interface `Iterable`. So, this statement will return a new instance of the static proxy class. This instance will be initialized with real subject instance of `Collection`. The figure 4 outlines the initialization of a new proxy with multiple subject interfaces.

```java
// Initializing the real subject
Collection c = new Collection();

// Initializing the proxy class and get an instance
Iterable pxy = new proxy<Iterable>(c);
```

**Figure 3.** Proxy initialization with a single subject interface.

```java
// Initializing the proxy class with multiple subject interfaces and get an instance
Iterable pxy = new proxy<Iterable, Serializable>(file);
```

**Figure 4.** Proxy initialization with multiple subject interfaces.

Adding a new keyword `proxy` ensures the traceability of this pattern in the source code of the application. Software engineers are able to see that an instance or a child class is linked to a generated proxy. The documentation traceability can also be improved, allowing the documentation generator to detect and document static proxy usages and redefinitions differently from other classes.

4.3 Proxy Classes: Static vs. Dynamic
The point of the notation `proxy<A,B>` is to denote a standard Java class automatically generated by the compiler. For compatibility we want that the generated class behave in the same way that proxy classes generated with JDP:

- Static proxy classes are public and not abstract. But unlike dynamic proxy class, they are not final and can be specialized (see section 5.)
- The unqualified name of a static proxy class is unspecified. However in the source code and in the messages of the compiler, the notation `proxy<A,B>` will be used to design the static proxy class based on the interfaces A and B.
- A static proxy class implements exactly the interfaces specified at its creation, in the same order.
- If a static proxy class implements a non-public interface, then the static proxy class will be defined in the same package as that interface. Otherwise, the package of a static proxy class is unspecified. It also means that all non-public interfaces must be in the same package.
- Two interfaces cannot have a method with the same name and parameter signature and different return type.

However, there is some big difference between static and dynamic proxy classes. The rest of this section exposes the details of the static proxy classes.

4.4 Proxy Constructor
Each static proxy class has one public constructor that takes one argument, a reference to the real subject. To instantiate a static proxy class, the programmer use the syntax `new proxy<I1,I2>(r)` where the list of interfaces is enclosed within angle brackets and the argument is the real subject.

If the real subject passed to the proxy initialization is `null`, a `NullPointerException` will be thrown at run-time.

To be statically safe, the real subject must be conform to each interface of the list. Eg. in `new proxy<I1,I2>(r)`, the static type of r must be a subtype of both I1 and I2. Unfortunately there is no way to express such a type constraint in the signature of the constructor\(^1\). Therefore the parameter is statically typed by the first interface of the list.

To ensure static typing safety, if the static proxy class is based on more than one interface, then the constructor of a static proxy class ensures that the dynamic type of the parameter is conform with every one of the interface. Therefore, at run-time, an `IllegalArgumentException` will be thrown for the instantiation of a static proxy class based of more that one interface if the real object is not conform to each one of the interfaces.

Relying on dynamic tests, even if it is only required for the rare proxies that implements more than one subject interface, is not something ideal. Therefore, the compiler should perform an additional verification on the static type on the argument passed to the constructor and warn the developer if this dynamic verification could fail. More concretely, the static type on the argument is compared to each interface of the list (excluding the first one that is already verified by the basic type system). If the argument is not conform to the interface I then the warning “Warning: the argument is not statically a I” is given to the programmer. Eventually, after verification, the developer can delete the warning with a `@SuppressWarning` annotation.\(^2\)

The figure 5 give an example for each possibility of a static proxy class instantiation.

4.5 Automatic Delegation
There is no `InvocationHandler` since the delegation to the real subject is automatically performed by the static proxy.

The static proxy class provides a concrete implementation of each method defined in the subject interfaces (and their superinterfaces). The provided implementation simply redirects the message to the instance of the real subject.

The number and the types of parameters used by each method is determined statically by the signature of each method in the subject.

To ensure static typing safety, if more that one interfaces have a method with the same name and parameter signature, then the return type must be the same or an error is given by the compiler. JDP require the same rule but is enforced dynamically at run-time by the `Proxy.getProxyClass` method that throws an `IllegalArgumentException`.

The provided implementation depends on the method return type. If the type is void, the method body consists only in a message redirection to the real subject with the given parameters. Otherwise, this call is preceded by the `return` keyword allowing the proxy to transmits the value returned by the real subject.

Exceptions declared by the provided implementation is the intersection of all exceptions of the methods declared in the superinterfaces.

---

\(^1\) Such a constrain requires some kind of union types, even if a proposal for Java exists [9].

\(^2\) One could want to turn the warning into a real error, but doing so will lead to problems in some rare corner cases. For instance, let an object be dynamically known to be both A and B (with `instanceof` for instance); this object can be safely casted to either a A or a B but not both at the same time, thus preventing the instantiation of a `proxy<A,B>` with this object as a real subject.
interface A {}
interface B {}
class C implements A {}
class D extends C implements B {}

C c = new C();
C cd = new D();
D d = new D();

// no warning and no run-time error
// d is statically and dynamically a A and a B
proxy<A, B> ab = new proxy<A, B>(d);

// a warning but no run-time error
// cd is dynamically a A and a B
// but statically we only know about the A
proxy<A, B> ab = new proxy<A, B>(cd);

// a warning and a run-time error
// c is not statically a B
// but neither dynamically
proxy<A, B> ab = new proxy<A, B>(c);

Figure 5. Example of safe and unsafe static proxy class instantiations.

4.6 Methods Inherited From Object

Since in Java, all classes inherit from Object class, primitive proxies will inherit all methods provided by Object and their standard implementations. Proxy must delegate all its methods to the real subject, but in some cases, the question about who is the actual recipient of the message must be asked? This problem is related to the self-problem as described by Bosch [1].

Dynamic proxies already offers a response to this problem. An invocation of the hashCode, equals or toString methods declared in java.lang.Object on a proxy instance will be encoded and dispatched to the invocation handler invoke method in the same manner as interface method invocations are encoded and dispatched, as described above. The declaring class of the Method object passed to invoke will be java.lang.Object. Other public methods of a proxy instance inherited from java.lang.Object are not overridden by a proxy class, so invocations of those methods behave like they do for instances of java.lang.Object[13].

To remain consistent with the specification of dynamic proxies, we maintain the same behavior. However, through the mechanism of specialization (Section 5), the developer is able to change the default implementation and choose if the message should be answered by the proxy or by the real subject.

4.7 ProxyType interface

Static proxy classes cannot extend java.lang.reflect.Proxy because there is no InvocationHandler field.

However, as described by Eugster[5], it is very useful to have a common supertype for all dynamic proxy types, be it for the mere purpose of testing whether an object is indeed a proxy. To that end, we reuse Eugster’s idea of the ProxyType interface and apply it to static proxies. The generated class still be a subtype of all subject interfaces and implements one more, the ProxyType.

4.8 Static Typing

Static proxies do not require an extension of the type system of the Java language.

The proxy construction can be used as a static type information. It refers to the actual hidden static proxy class. This is also the static type of a new proxy expression.

However, the usage of such a type is not really useful in programs since the proxy behavior must be represented by an interface [6], whenever a proxy is expected, the subject interface type should be used instead.

The use of the proxy construction as a type may, however, be useful when methods are defined in two different subject interfaces. An example of a two references use is given in figure 6.

interface A {
    public void foo();
}

interface B {
    public void bar();
}

...proxy<A, B> ab = new proxy<A, B>(realSubject);
ab.foo(); // Access to subject A methods
ab.bar(); // Access to subject B methods

Figure 6. Using a proxy typed by two different interfaces.

Except for the latter case, there is in practice no real advantage to allow the usage of a proxy construction of a class. Forbidding the use of the proxy construction in a static type may be a reasonable option. Again, union types applied to Java, could be a better way to deal with proxies based on multiple interfaces.

5. Specializing Static Proxy Classes

We want to reduce to a minimum the manual implementation work needed to use the proxy class. We generate all valueless code like message forwarding. The Budinsky [2] and Sunyé [14] approaches shows that implementation trade-offs must be left to the developer, the only one able to write the specific code for the specific situation.

To add a new method or replace the naive implementation, the developer simply has to inherit the proxy class by a more sophisticated one. As seen in FACE [10], specialization of basic proxy enhances static type checking of the software engineer trade-offs and improves traceability.

We think that specializing the proxy class to refine the proxy behaviors is a more elegant way than using reflection to modify the proxy object.

5.1 Syntax

The same proxy keyword can be used for static proxy class specialization as shown in figure 7. To inherit from a static proxy class, the proxy keyword is used in the inheritance clause of the class followed by the related interface. This indicates to the compiler to refer to the proxy class rather than to the subject interface. The last line gives an example the instantiation of such a subclass, by the classic new keyword like other classes.

With the static proxies the developer is able to change the behavior of methods inherited from Object. Figure 8 shows an example of a redefinition of the method toString.

5.2 Real Subject References

In a subclass, the real subject must be given to the proxy class through its constructor via a super call.

Because the reference to the real subject stored in the static proxy class is private, the reference is not visible in the subclasses. To use the reference locally in a subclass, the developer has to use local field. Figure 8 shows an example with the realSubject attribute definition.
we have the following:

• Because the super call from the constructor suffers from same
  problems (and solutions) than discussed in 4.4, there will be a
  warning.

• The same real subject is stored in a different field (one field per
  interface). Each field will access the methods defined by one or
  the other interface.

A cast is used to transform the static type of the real subject in
the constructor. However, these casts cannot fail since the super
constructor did not throw a IllegalArgumentException.

5.3 Changing the Proxy Constructor Behavior

The default behavior of the static proxy instantiation is to create a
new instance each time this constructor is called because the new
keyword indicates that a new object is instantiated and that is the
simplest and intuitive behavior for the developer.

However, many instances of proxies based on the same interface
and the same real subject instance are useless since all messages
will be forwarded to the same instance of the same real subject.
In practice, if the software engineer needs to save space, he can
implement himself a specific singleton mechanism which use a
private redefinition of the default constructor. Example is given in
figure 10.

5.4 Preserving Proxy Encapsulation

As Gamma shows in [6], the proxy design pattern can also be con-
sidered a protective mechanism to limit access to some services
provided by the real subject. That is proxy encapsulation. Allowing
software engineer to redefine the behavior by specializing the prim-
itive proxy class can lead to an unwanted encapsulation break. In
some cases, this may be the primary implementation itself, which
destroys the encapsulation. This is also related to self-problem.

Figure 7. Specialization of the generated proxy class for interfaces
Resource and Serializable.

class CountAccessProxy extends proxy <Iterable> {
    private int count;
    public CountAccessProxy(Iterable i) {
        super(i);
        this.count = 0;
    }
    @Override
    public Iterator iterator() {
        this.count++;
        return super.iterator();
    }
    // Redefinition use
    Iterator pxy = new CountAccessProxy(c);
}

Figure 8. Redefinition of the proxy toString method default be-
behavior.

When the proxy is based on a unique subject interface, a simple
field typed by the subject interface is enough to store the real
subject in the class. Note that in some cases, the reference to the real
subject is not necessary as shown by figure 7 where the reference
to the real subject is directly passed to super without being used
or stored locally.

If the proxy is based on more than one subject interface then, the
static type of the real subject is more difficult to determine5. This
yields to various questions: what is the static type of the parameter
of the constructor? How to call the super constructor? What is the
type of the local field used to store the real subject?

The static type of the parameter of the constructor can be set to
a class (or an interface) that specialize all the interfaces of the
static proxy class. While this reduce the usage of the subclass, this
solution solves the two other problems.

Alternatively, the solution of the static proxy class can be reused
and the static type of the parameter of the constructor can be set to
first interface of the list. With this solution, illustrated by figure 9,
we have the following:

- The documentation of the constructor must state clearly that
  the argument must also be dynamically conform to each other
  interface.

- Because the super call from the constructor suffers from same
  problems (and solutions) than discussed in 4.4, there will be a
  warning.

- The same real subject is stored in a different field (one field per
  interface). Each field will access the methods defined by one or
  the other interface.

A cast is used to transform the static type of the real subject in
the constructor. However, these casts cannot fail since the super
constructor did not throw a IllegalArgumentException.

class RefinedProxy extends proxy <A, B> {
    private A a; // The real subject seen as a A
    private B b; // The real subject seen as a B

    /** doc: rs must also be a B */
    public RefinedProxy(A rs) {
        @SuppressWarning // because rs is not
        super(rs); // statically a B
        this.a = a;
        this.b = (B)a; // this cast cannot fail
    }
    @Override
    public void foo() {
        a.foo();
    }
    @Override
    public void bar() {
        b.bar();
    }
}

Figure 9. Defining the real subject reference with a multiple inter-
face based proxy.

---

5 Once again, union types...
A Static Alternative To Java Dynamic Proxies.

class SingletonProxy extends proxy<Something> {
    static private Map<Something, SingletonProxy> proxies = new HashMap<Something, SingletonProxy>();
    
    static final SingletonProxy getInstance(Something realSubject) {
        if (proxies.containsKey(realSubject)) {
            return proxies.get(realSubject);
        } else {
            SingletonProxy proxy = new SingletonProxy(realSubject);
            proxies.put(realSubject, proxy);
            return proxy;
        }
    }

    private SingletonProxy(Something obj) {
        super(obj);
    }
}

Figure 10. Example of singleton proxy mechanism.

Figure 11. Encapsulation break by returned values

// Get a new proxy instance
Node px = new proxy<Node>() { new XMLNode() };

// Here, static type of 'proxy' still 'Node'
// but instance returned by prev() is
// dynamically typed 'XMLNode'
pxy = px.y.next().prev();

// The cast exhibits the encapsulation break
XMLNode rs = (XMLNode)pxy;

// Access to hidden method setValue()
rs.setValue("exploit");

Figure 12. Exploit of the encapsulation break

Implementation Overhead Unlike the manual static approach, in our proposal the programmer is not required to implement the trivial part of the proxies. In comparison with JDP, the implementation is less complex.

In basic cases where the static proxy class with systematic delegation to the real subject is enough, no code at all is required (whereas JDP require the implementation of a invocation handler even for the simple cases). With more complex proxies, our approach is straightforward since the developer simply has to implement a subclass and redefine the methods he wants (the developer is not required to understand introspection in Java).

Reusability There is a lot of possibility of reusing through the specialization of static proxy classes.

Code Traceability The proxy keyword clearly identify the code that deals with proxies. At run-time, the interface ProxyType can also be used to discriminate between proxies and non-proxies.

Trade-offs Facilities While the basic static proxy class behavior is imposed, a lot of variations and customizations is possible through specialization.

Static Reliability One of the main objective of our proposal. Static reliability is enforced mostly by the standard Java static type system. What remains is enforced by warnings in the compiler.

Code Evolution Static proxy classes are automatic but cannot be dissynchronized with the interfaces since no Java code is actually generated for the proxy. Subclass of static proxy classes will either continue to work or will refuse to compile: run-time surprises are more unlikely than with JDP.

Performance Generated code is equivalent to a manual proxy implementation; therefore both should have the same run-time performance.

6. Static Proxies in OpenJDK 7

To verify the usability of our method and the gain provided to the developer in terms of writing less code, we chose to rewrite a portion of an existing library. We therefore introduce the static proxies in the OpenJDK 7 javac compiler. No changes to the virtual machine is made. This section discusses the implementation detail of our approach.

The static proxy implementation with OpenJDK 7 is available online.4

6.1 Proxy Class Creation

The compiler does not need to generate a new proxy class each time a static proxy class is refereed in the code (in a new or an extends) with the same list of interfaces. The Java compiler that implements static proxy classes should keep a cache of generated proxy classes, keyed by their corresponding interface list. This is the same behavior as JDP except it is done at compile-time.

As specified in section 4.3. If a static proxy class implements a non-public interface, then the static proxy class will be defined in the same package as that interface. In all other cases, we use the same package than JDP to store the new proxy class in $Proxy.

6.2 Proxy Class Naming

As in dynamic proxies specification, the unqualified name of a proxy class is unspecified for developer. We still need to name it internally. For this, we name by "$Proxy" concatenated followed by the qualified name of each subject interface separated by a "$".

4 Because of the double-blind reviewing we are not sure if we can provide the url. However, the url will be available in the final version of the paper.
Dots characters figuring in qualified name space are replaced by a single "$". The space of class names that begin with the string "$Proxy" is, however, to be reserved for proxy classes [13].

For example:
- $Proxy$java$io$Serializable refers to proxy<Serializable>
- $Proxy$java$io$Serializable$java$lang$Runnable refers to proxy<Serializable, Runnable>

6.3 Proxy Class Constructor

Each proxy class has one public constructor that takes one argument, the real subject. We show in section 4.3 that in an ideal world this argument must be typed by the union type of all the subject interfaces.

To simulate the types union representing the type of the real subject, we use a combination of static and dynamic type checking.

The static type of the argument of the constructor is the first interface of the list of interfaces. Thus, a static verification is done by the compiler to ensure that the real subject complies with this constraint.

To avoid type errors when calling methods of other interfaces we need to ensure that the real subject is subtype of the other interfaces. For this, we introduce dynamic type tests in the constructor to ensure that the real subject is subtype of all other interfaces. Figure 13 gives an example of dynamic type checks added to the constructor.

In the specification, the unqualified name of a static proxy class is unspecified for the developer. Thus, the compiler has to name it internally in a way that prevents conflicts. In our implementation, the name is "$Proxy" concatenated to the qualified name of each subject interface separated by a "$". Dot characters figuring in qualified name space are replaced by a single "$". The space of class names that begin with the string "$Proxy" is, however, to be reserved for proxy classes [13].

As seen in 5.2 the references to the real subject are difficult to type statically. In the implementation of the proxy class in Java, we have chosen to create a field for the real subject for each interface. All fields contain the same reference to the same instance of the real subject but with different static types.

When generating the methods delegation of an interface, it the field statically typed like the interface that is used to make the call. In the example given in figure 13 the fields realSubject1 is used to delegate methods of Readable while realSubject2 is used to delegate methods of Runnable.

Fields are named by a sequence number prefixed by "realSubject".

6.4 Example of Generated Static Proxy Class

Figure 13 gives a full example of Java pseudo-code for the proxy class proxy<Readable, Runnable> generated at compile time.

In fact no Java code is generated since we have used the OpenJDK framework to directly produce equivalent .class files (byte-code).

7. Real Case Example

In this section we compare the replacement of a manual static implementation of the proxy pattern with our proposal. The real example is the Wrapper implementations\(^3\) of Java standard library collections that are proxy around standard collection. All these wrappers are inner classes of java.util.Collections.

3\(^3\)\url{http://download.oracle.com/javase/6/docs/technotes/guides/collections/reference.html}

8 2012/4/14

class $Proxy$java$lang$Readable$java$lang$Runnable
implements Readable, Runnable {
    private Readable realSubject1;
    private Runnable realSubject2;

    public $Proxy$java$lang$Readable$java$lang$Runnable
    (Readable realSubject) {
        this.realSubject1 = realSubject;
        try {
            this.realSubject2 = (Runnable) realSubject;
        } catch (ClassCastException e) {
            throws new IllegalArgumentException(e.getMessage());
        }

        @Override
        public int read(CharBuffer cb) {
            return this.realSubject1.read(cb);
        }

        @Override
        public void run() {
            this.realSubject2.run();
        }
    }
}

Figure 13. Example of proxy class $Proxy<Readable, Runnable$ as generated at compile time for a single interface Subject.

The class Collection provides functionality-enhancing implementations of collections for use with other implementations. These implementations can be grouped in three categories:

Collections.unmodifiableInterface Return an unmodifiable view of a specified collection that throws an UnsupportedOperationException if the user attempts to modify it.

Collections.synchronizedInterface Return a synchronized collection that is backed by the specified (typically unsynchronized) collection. As long as all accesses to the backing collection are through the returned collection, thread-safety is guaranteed.

Collections.checkedInterface Return a dynamically typesafe view of the specified collection, which throws a ClassCastException if a client attempts to add an element of the wrong type. The generics mechanism in the language provides compile-time (static) type checking, but it is possible to defeat this mechanism. Dynamically typesafe views eliminate this possibility entirely.

Without modifying the current class hierarchy of Java collections library, our static proxies can be introduced in nine classes.

To introduce the primitive proxy construct in those classes, we replace the implements clause of the subject interface by an extend of a static proxy class of the proxyed interfaces, then we delete all method whose implementation simply forwards messages to the real subject. Figure 14 gives an overview of the original UnmodifiableCollection class. Figure 15 presents the modified class.

Detailed results for all nine classes modified are given in table 16. Results show that using static proxies can reduce the number of defined methods by 35%.

The static proxies appear as a complementary feature to JDP. In some cases, where the use of reflection is not justified, the static
A Static Alternative To Java Dynamic Proxies.

8. Related Work

Many studies have been completed to handle patterns implementation problems like manual implementation, code and documentation traceability or maintenance.

Tools exist to automatically generate patterns source code and sometimes also the related documentation [2, 14]. These approaches have a difficulty in managing the integration problems of the pattern and its evolution. This seems to be directly related to the philosophy of external tools that suffer from too much independence from language implementation. More efficient approaches seem to be located closer to the application source code.

static class UnmodifiableCollection \<E\>
impliments Collection \<E\>, Serializable {
final Collection \(<? extends E>\) c;
UnmodifiableCollection (Collection \(<? extends E>\) c) {
    if (c==null)
        throw new NullPointerException();
    this.c = c;
}
public int size () {
    return c.size();
}
public boolean contains (Object o) {
    return c.contains (o);
}
... public boolean add (E e) {
    throw new UnsupportedOperationException ();
}
public boolean remove (Object o) {
    throw new UnsupportedOperationException();
}
...}
Figure 14. Original UnmodifiableCollection class.

static class UnmodifiableCollection \<E\>
extends proxy \<Collection \<E\>> implements Serializable {
UnmodifiableCollection (Collection \(<? extends E>\) c) {
    super (c);
    if (c==null)
        throw new NullPointerException();
} // no need to reimplement the simple forward methods
public boolean add (E e) {
    throw new UnsupportedOperationException();
}
public boolean remove (Object o) {
    throw new UnsupportedOperationException();
}
...}
Figure 15. Modified UnmodifiableCollection that extends the static proxy class for the collection interface

proxies provide an interesting and convenient answer that allows the developer to save time.

Figure 16. Number of methods in each class before and after the use of static proxies.

8. Related Work

Many studies have been completed to handle patterns implementation problems like manual implementation, code and documentation traceability or maintenance.

Tools exist to automatically generate patterns source code and sometimes also the related documentation [2, 14]. These approaches have a difficulty in managing the integration problems of the pattern and its evolution. This seems to be directly related to the philosophy of external tools that suffer from too much independence from language implementation. More efficient approaches seem to be located closer to the application source code.

Closer the code, there are some static approaches like proxy generation by macro-commands [12] or using frameworks [10]. All of them attempt to abstract the proxy at code level in order to apply static type verifications. It shows that working in a static world can limit the expressiveness of the solution. LayOM [1] (for C++) and OpenJava [15] (for Java) give an elegant response to expressiveness by introducing a meta layer describing the pattern behavior.

As with Java Dynamic Proxies [5, 13], reflection is among the preferred the way of run-time generation of proxies. Such a meta-programming is used in many languages like Javascript [16], Eiffel [4], or Ruby [11]. Reflection allows automated changes into the object model and limit the code to be written manually. However, everything is done dynamically at run-time, it involves a performance overhead and is more difficult to apprehend for programmers.

Eugster [5] offers a version of dynamic proxies for use with subjects represented by classes. The problem of initialization is managed through an initializer containing only a public constructor without arguments used to initialize the proxy instance. Attribute access is solved by using getters and setters added implicitly by reflection.

Hannemann [8] provides a way to address the distribution of a preoccupation through a set of classes in Java using aspects. This approach is more effective with complicated patterns involving several classes. It introduces an additional level of complexity to the simple case of proxy.

9. Conclusion

In this paper we propose a static alternative to proxy generation in Java and compare it to the Java Dynamic Proxies solution.

The Dynamic Proxies use reflection to generate the proxy at run-time. While useful in cases where dynamic approach is the only way to build the proxy, it has three main drawbacks: (i) lack of static guarantees, (ii) complex usage requiring introspection, and (iii) performance overhead.

We propose a static alternative to the generation of proxies. We keep the solutions introduced by Java dynamic proxies and adapt them to the static world. Our proposal is consistent with the dynamic proxies and do not requires an extension of the type system of the Java language.

Rather than using reflection to generate the proxy class, we rely on the services of the compiler. The proxy is introduced into the model of the application at compile time. The use of the class loader or the invocation handler is no longer necessary. The type constraints checking is mainly delegated to the compiler that allows errors to be given at compile-time instead of run-time.

The static approach is highly dependent on the expressiveness of static typing of Java. Particularly in regard to the union type of all the subject interfaces and references to a class that will be generated
at compile time. We propose some alternatives to overcome the limitations of the Java type system for proxies.

To test our approach, we implemented it in the OpenJDK 7 javac compiler. The evaluation is made using wrapper implementation classes of Java standard Collection library. The results show that static proxies, can reduce the number of method by 35%. Static proxies appear to be type-safe and fast to use. Trade-offs can be implement easily and safely by specialization in a straightforward way.

To conclude on this approach we note that the Java type system lacks expressiveness to represent some concept statically and this is the case with the proxy. Types unions in this case the only limitation.

In future work, include combination of static proxies with type union in Java. Also, an extension of the capabilities of static proxies to base on subject class can be considered.

References