Proxy chains (also known as: Delegator pattern)

The proxy chain is best explained with the situation it was made for. Suggest a really complex source generator with varying options. For example, a child element of an object can have multiplicity zero, one or more: In the latter case the child element will most probably be represented by a list. Quite necessarily the source generator will sonner or later be bloated with statements like

```
if (child.isMultiple()) {
    // Do something, working with a List
    ...
} else {
    // Do something else, working with a single element
}
```

Even worse, the code will in both cases look quite similar: As soon as the list item is fetched and casted to a proper type, this is just like working with a single item. As time goes by, more and more similar options come into play: Localization, varying datatypes, and so on. Soon you start to think of better solutions.

The first attempt is typically subclassing. For example, if you have a source generator for the case multiplicity zero or one, it can well be expanded to the general case, with multiplicity two or more being handled in the subclass. Localization is handled in the next subclass, and so on. Your code looks much better now and becomes maintainable again.

However, time goes by, and more options are added to the source generator: Transaction handling, encoding issues, and so on. Soon you find yourself in trouble again: The subclassing approach doesn't work so well anymore, because in a subclass chain E -> D -> C -> B -> A (where B is a subclass of A, C is a subclass of B, and so on) you sometimes want all of them, but sometimes only D -> C -> A or E -> A. At that point of thinking a new approach was born: The event chain.

The idea is replacing the subclasses with an event interface. For example, suggest that the source generator might contain code like the following:

```
public class FooGeneratorImpl implements FooGenerator {
    public JavaSource getFooClass() {
        JavaSource js = factory.newJavaSource(fooClassName, "public");
```

```
            getXMethod(js);
            getYMethod(js);
            getZMethod(js);
        }
        ...
    }
```

The corresponding interface might look like this:

```
    public interface FooGenerator {
        public JavaSource getFooClass();
        public JavaMethod getXMethod(JavaSource js);
        public JavaMethod getYMethod(JavaSource js);
        public JavaMethod getZMethod(JavaSource js);
    }
```

If you take this methods as events, then you might well write a default class A implementing the interface. The other classes are implemented as subclasses of an automatically generated proxy class. For example, the class B might just add another method to the `Foo`. This might look like the following:

```
    public class B extends FooGeneratorProxy {
       // Override only the getFooClass() method, all other methods are
       // passed to the base generator by the proxy.
       public JavaSource getFooClass() {
           JavaSource result = super.getFooClass();
           JavaMethod bMethod = result.newJavaMethod("bMethod", "void", "public");
           return result;
       }
    }
```

Likewise, the C class might change the interface of method X, and so on. Any feature is implemented by a single class, which you can optionally add to the chain (turning the feature on) or remove from the chain (turning the feature off).

However, there's still a problem left: When you are inside A (the topmost class) and do a `getXMethod()`, then you call your own class and not the chains bottom. This problem is fixed by the following design:

Controller					Implements FooGenerator
E	D	C	B	A	Implements ChainedFooGenerator

The `ChainedFooGenerator` is exactly matching the `FooGenerator` interface, the exception being an additional parameter `FooGenerator pController` in all methods.

Proxy chains (also known as: Delegator pattern)

The FooGenerator interface can be created automatically, also an implementation of FooGenerator calling the first element in the chain of `ChainedFooGenerator` implementations. The manually created classes have to be changed slightly, here's the updated `FooGeneratorImpl`:

```
public class FooGeneratorImpl implements ChainedFooGenerator {
    public JavaSource getFooClass(FooGenerator pController) {
        JavaSource js = factory.newJavaSource(fooClassName, "public");
        pController.getXMethod(js);
        pController.getYMethod(js);
        pController.getZMethod(js);
    }
    ...
}
```

Likewise, here is the updated B class:

```
public class B extends ChainedFooGeneratorProxy {
   // Override only the getFooClass() method, all other methods are
   // passed to the base generator by the proxy.
   public JavaSource getFooClass(FooGenerator pController) {
       JavaSource result = super.getFooClass(pController);
       JavaMethod bMethod = result.newJavaMethod("bMethod", "void", "public");
       return result;
   }
}
```

The proxy chain pattern is implemented by the ChainGenerator. From within Ant, it looks like the following:

```
<chainGenerator destDir="src">
         <chain
            controllerInterfaceName="com.dcx.sein.dbtk.generator.javasg.IModelSG"
            chainInterfaceName="com.dcx.sein.dbtk.generator.javasg.auto.IChainedMod
            proxyClassName="com.dcx.sein.dbtk.generator.javasg.auto.ChainedModelSGP
            implementationClassName="com.dcx.sein.dbtk.generator.javasg.auto.ModelS
         <chain
            controllerInterfaceName="com.dcx.sein.dbtk.generator.javasg.IObjectSG"
            chainInterfaceName="com.dcx.sein.dbtk.generator.javasg.auto.IChainedObj
            proxyClassName="com.dcx.sein.dbtk.generator.javasg.auto.ChainedObjectSG
            implementationClassName="com.dcx.sein.dbtk.generator.javasg.auto.Object
</chainGenerator>
```

The `controllerInterfaceName` is the name of the basic interface. This is what you actually want to use from the outside. The controller interface must be available as a compiled class, because it is inspected with Java reflection. The other classes are generated: The `chainInterfaceName` is the interface being implemented by the manually written classes. The `proxyClassName` is an automatically generated implementation of the

`chainInterface`, which passes all events to the next element in the chain. And the `implementationClassName` is an also automatically generated implementation of `controllerInterface`, that works internally by passing all events to the first element in the chain.