# Demonstration of Factory - A Java Extension for Generative Programming

Christof Lutteroth

Institute of Computer Science
Freie Universität Berlin
Takustr.9, 14195 Berlin, Germany
lutterot@inf.fu-berlin.de

**Abstract.** Factory is an extension of Java which provides a template-based reflection mechanism for generative programming. Java classes can be parameterized by types, and the structure of the classes can be described dependent on these type parameters. Factory can address a wide range of applications and save programmers a lot of work. It is designed to integrate seamlessly with Java, to be intuitive for the user, extensible, and safe.

## 1 Introduction

Generative programming is about the idea to automate parts of the software development process. It is a paradigm that tries to bring the development of software onto a new level of abstraction and provide new means of reuse. Usually, there are programming tasks in software projects which are so regular that we can make the computer do them for us by programming a generator. A common example is a compiler, which generates the representation of a program in one language from its representation in another. Of course, we could do this translation by hand, but with the help of the compiler a lot of time is saved and we are able to focus on the aspects of our program on a level that is much more abstract than machine language. Programming languages, which are often implemented as compilers, can make a big difference to the programmer with the level of abstraction they provide.

There are many such tasks for which technologies of generative programming are used. For example, there are compiler generators, tools for the generation of database interfaces (e.g., [3]) and stub-generators (e.g. the Java *rmic* tool [12]). But it is not always an external tool that performs generation: Many programming languages have inbuilt features for generative programming, even though they are usually not seen from that point of view. A macro mechanism, for example, like the C preprocessor, is a generative feature, and also some features of object-oriented languages, like inheritance. In all such cases, the information given in the source-code is used to generate, according to a well-specified pattern, code with properties that are not expressed directly in the source.

Besides the widespread traditional mechanisms, like macros and inheritance, there are also more advanced ones. Nowadays, many programming languages incorporate parametric polymorphism [4], also known as generic types, and introspective access to runtime objects (as, for example, provided by the Java reflection API [12]). Then, there exist more complex mechanisms, reflective architectures [6] that allow many aspects of a program to be changed at compile-time or even at runtime. For a general account on such technologies, see [7] or [5].

Factory [9] is a generative programming concept for problems for which parametric polymorphism without reflection, as known from C++ [11] and other languages [1], is not sufficient. The name "Factory" points out its generative capabilities on the level of metaprogramming and should not be confused with the factory design pattern. While the factory design-pattern refers to a class that creates objects, our factories are template-like entities that create classes. One could say that Factory takes the design pattern from the level of object generation to the metalevel of class generation.

Parametric polymorphism without reflection is used, e.g, for building type-safe containers, like lists or hash tables. But many important applications for generative programming require more powerful mechanisms. Examples that cannot be solved with traditional parametric polymorphism include those in which not only types are substituted but new signatures and code are created. A generator that yields EJB-conformant [10] wrappers for an arbitrary interface would be such an example. For each method of a given interface, the wrapper would have to generate a method with a different signature, dependent on the signature of the original method. Other examples include the generation of test suites for given classes, of a database interface, error handlers, stubs, etc. A generative approach to address these examples needs *introspection*, i.e., queries to the metamodel of the source-code that is dealt with, in order to explore all methods, as well as *intercession*, i.e., modifications of the metamodel, in order to create the new methods and their signatures (see also [6]).

Factory does support this kind of reflection, with a focus on *static reflection*, i.e., reflection done at compile-time. This is because in most practical examples, compile-time reflection seems to be sufficient. Runtime reflection is only needed in special cases, e.g., for hot deployment enabled containers in adaptive systems. Nevertheless, Factory is also capable of runtime reflection, but yet, compile-time reflection is easier to use.

Factory has been designed to provide a particularly strong notion of safety, *generator-type-safety*, by enabling certain analysis techniques of the Factory source-code representation at definition-time. Generator-type-safety guarantees the type-safety of all classes that can possibly be generated by a Factory generator. It can be verified with help of a type system [2]. Furthermore, Factory can determine if a generator terminates always.

## 2 The Factory Language

Factory has its own language that embeds a specialized XML-like syntax for compile-time generation into the standard syntax of Java 1.4. It uses the template approach, so the programmer only needs to use Factory specific syntax whenever he wants to make use of its generative capabilities. Apart from that, programming is the same as in Java. However, the concept of Factory is not restricted to Java, and one day, it might be a good choice to integrate the compile-time part and the runtime part of the Factory syntax into a homogeneous, language independent abstract syntax.

The unit of generation and compilation is a *factory*, a file which contains exactly one class or interface definition (excluding inner interfaces/classes). It can be parameterized, use the Factory syntax of generative control constructs and generative terms, and invoke any number of other factories. But since the syntax is, apart from these Factory-specific extensions, that of a normal Java source file, all Java source files that contain only one class or interface (inner ones excluded) are valid factories.

### 2.1 Generator Variables

Generator variables are the variables used at generation-time, as opposed to the normal Java variables which are used at runtime. They are similar to normal Java variables, but they contain only object values. This is not a restriction, since the Factory system performs, when calling methods, casts to and from primitive types automatically. Furthermore, not all of these variables have an explicit type bound, but simply accept any object as value. Since the Factory generation system works with factory terms, which are functional, it is not possible to directly assign a value to a variable after it has been declared. There are several ways in which generator variables can be introduced into a factory: by declaring a Factory parameter with the `<param>` tag or by using the `<for>` or `<let>` constructs.

### 2.2 Factory Terms

Factory terms are a functional notation with which Java classes and other factories can be accessed in a safely restricted way. Usually, those terms are used to introspect the type parameters of the respective factory and to extract or construct the information that is needed for intercession.

It is one of the basic decisions in the design of Factory not to create a new metamodel for reflection but rather to integrate Factory seamlessly into the existing metaobject protocol of Java. This makes it a lot easier for people with a knowledge of the Java reflection API to use Factory. Furthermore, the integration with Java makes it easy to use and extend Factory with all the possibilities that Java offers. It is possible to access the standard classes as well as self-made ones, given that all the fields, methods and constructors that should be accessible are registered with the Factory system. Only classes that are considered safe, i.e., that terminate and have no harmful side effects, should be registered. The syntax of a Factory term is defined as follows:

$\underline{\text{term}}$: ( $\underline{\text{constant}}$

```
          |   variable
          |   get
          |   application )
```

Internally, those terms work with objects, but any method that specifies parameters with primitive types can be used with objects of the corresponding class types. Also, if a method returns a value of a primitive type, it is internally converted into an object of the corresponding class type.

Constant literals can be created with constant terms that use the `<const>` tag. With this tag, objects for all the primitive Java types can be created, simply by using the respective literal standard notation. Also instances of metaclass `class` can be created by specifying the fully qualified class name.

Generator variables can be accessed with `<var>`:

```
variable:   <var> IDENT </var>
```

Member variables of Java classes can be accessed with `<get>`. If the first tag in the body is another factory term, a member variable of the returned object is accessed; if it is a `<class>` tag, a static member variable of the named class is accessed.

```
get:  <get>
      (   term
      |   <class> CLASS_IDENT </class> )
      <field> VAR_IDENT </field>
      </get>
```

Applications are done with an `<apply>` tag. If the first tag in the body is a term, a method is invoked on the object returned by that term. If it is a `<class>` tag followed by a `<method>` tag, the respective static method of the named class is invoked. If there is just the `<class>` tag, a constructor for the named class is invoked. If the `<factory>` is the first tag in the body of `<apply>`, a factory generator is applied, which returns the `Class` object for the generated class or interface. All applications may give arguments in the form of other terms in the `<args>` tag. If there is no argument, `<args>` can be left out.

```
application:  <apply>
              (   term
                  <method> METHOD_IDENT </method>
              |   <class> CLASS_IDENT </class>
                  (<method> METHOD_IDENT </method>)?
              |   <factory> FACTORY_IDENT </factory> )
              (<args> (term)+ </args> )?
              </apply>
```

### 2.3 Intercession with Factory Terms

Factory terms are placed at certain positions into standard Java source-code in order to perform intercession, i.e., to make the result of the term generate an element of the Java syntax. The terms are placeholders for Java syntax, which is filled in at generation-time.

Factory terms are only allowed at certain syntactical positions, and the position depends on the type of the term. In other words, if we want to generate a certain syntax element at a certain position, we have to use a term that evaluates to an object that models that syntax element correctly. The following table lists valid return types of Factory terms and positions where those terms can be used. In the second column, at the place of the term, we inserted the name of the class of which an object must be returned by the term.

| Type | Possible Positions |
|---|---|
| `Class` | Instead of types:<br>`Class` x;<br>`Class` myMethod() ...<br>int myMethod(`Class` x) ...<br>x = (`Class`) y; |
| `String` | Instead of most identifiers:<br>class `String` ...<br>int `String`;<br>x = `String` + 1;<br>x = `String` (1); |
| `Package` | package `Package` ; |
| `Integer` | Instead of modifiers:<br>`Integer` class C { ... }<br>`Integer` int x;<br>`Integer` int myMethod(...) { ... } |
| `Method` | Instead of method head:<br>`Method` { ... } |
| `Class[]` | Instead of parameter or argument list:<br>public int myMethod `Class[]` { ... }<br>x = myMethod `Class[]`; |
| `Argument[]` | Instead of argument list:<br>x = myMethod `Argument[]`; |

### 2.4 Partial Evaluation

Factory terms can also be used in order to perform a simple partial evaluation. This optimization allows to do computations at generation-time and insert the result into the generated class.

```
class Calculator {
  final double pi =
    <literal>
      <apply>
        <class> myPackage.myClass </class>
        <method> calcPi </method>
      </apply>
    </literal>;
  ...
}
```

The `<literal>` tag can be used in Java expressions and must contain a term that evaluates to an object corresponding to a primitive Java type or `String` – those types, for which the Java syntax defines literals.

## 2.5 Control Constructs

In addition to terms, Factory provides control constructs: an `<if>`-tag for conditional generation, a `<for>`-tag for iterative generation, and a `<let>`-tag for assigning the value of a term to a new generator variable. Each of these constructs is available in two variants: one variant that can be used in place of a Java statement, e.g., in a method body, to generate statements, and one variant that can be used in place of a field to generate member variables and methods. Consequently, the nonterminal symbol element in the following rules can be either a statement or a field, depending on where the construct is placed.

The `<for>` allows to iterate over the elements of any array object. The array object must be given by the term after `<var>`. During generation, the fields or statements in the body are generated for each element in the array, and in each iteration, the respective element can be accessed in the generator variable declared in the `<var>` tag.

for:
`<for>` `<var>` IDENT `</var>` term
`<body>` ( element )$^*$ `</body>`
`</for>`

The `<if>` allows conditional generation. The term after `<if>` must evaluate to an object of type `Boolean`, and if its value is true, the field(s)/statement(s) in the `<then>` tag are generated, otherwise the ones in the `<else>` tag. The `<else>` is optional.

if:
`<if>` term
`<then>` ( element )$^*$ `</then>`
( `<else>` ( element )$^*$ `</else>` )$^?$
`</if>`

The let-construct allows to use a new generator variable in place of a term. The construct declares the variable and assigns it the value of the term, which can then be used in the `<body>`.

let:
```
<let> <var> IDENT </var> term
<body> ( element )* </body>
</let>
```

The `<let>` can be seen as a special case of the `<for>` because it can be reduced to a loop that iterates over a single-element array.

## 3 Example for Compile-Time Reflection: Generating Setters

For an actual type parameter `Person`, the following factory `Setters` generates a class `PersonWithSetters` that extends class `Person`. `PersonWithSetters` overrides each public member variable of `Person` with a private member variable of the same type and name and declares a setter-method for each of it, similar to the convention of (non-enterprise-) JavaBeans.

This is more an academic example that demonstrates the reflexion capabilities of Factory. In real world applications, getter- and setter-methods are usually used in order to do something more than just reading or writing a single member variable, like notifying other objects when a value is changed.

```
<param>
    <bound> java.lang.Object </bound>
    <var> T </var>
</param>

public class
    <apply>
        <apply>
            <var> T </var>
            <method> getName </method>
        </apply>
        <method> concat </method>
        <args> <const> "WithSetters" </const> </args>
    </apply>
extends <var> T </var> {
    <for> <var> I </var>
    <apply> <var> T </var>
        <method> getFields </method>
    </apply> <body>
        <let> <var> FT </var>
        <apply> <var> I </var>
            <method> getType </method>
        </apply> <body>
            <let> <var> FN </var>
            <apply> <var> I </var>
```

```
                <method> getName </method>
            </apply> <body>
                private <var> FT </var> <var> FN </var>;

                public void
                <apply>
                    <const> "set" </const>
                    <method> concat </method>
                    <args> <var> FN </var> </args>
                </apply>
                (<var> FT </var> value) {
                    this.<var> FN </var> = value;
                }
        </body> </let> </body> </let>
    </body> </for>
}
```

The following factory applies factory `Setters` to class `Person`, instantiates an object of the resulting class `PersonWithSetters` and uses its setter-method for its private member variable `plz`. If we uncomment line 12, which tries to access `plz` directly, we would not be able to compile it because a private member variable cannot be accessed outside of its class.

```
 1  class SettersTest {
 2      public static void main(String argv[]) {
 3          <let> <var> T </var>
 4          <apply>
 5              <factory> Setters </factory>
 6              <args> <const> Person </const> </args>
 7          </apply>
 8          <body>
 9              <var> T </var> p = new <var> T </var>();
10              p.setPlz(1);
11
12              // p.plz++;
13          </body> </let>
14      }
15  }
```

## 4 Runtime Reflection

Since Factory is itself written in Java, it can easily be used from within other Java classes. This means that the generation process done by a factory can be invoked at runtime; and since Java supports dynamic class loading and reflective access to classes, the generated classes can be used straight away. This example demonstrates how Factory can be used for runtime reflection, as it is useful in hot-deployment enabled environments. It would enable components, e.g., GUI components, to adapt dynamically to system changes, upgrades and extensions, providing a high degree of flexibility and tolerance. A type checker for generator-type-safety can statically ensure the dynamic safety of such adaptive components because it would assure that no adaption would bring the component into an erroneous state.

The following Java source-code snippet applies the factory `EditFrame` dynamically to the `Class` object in variable `t`: it creates an instance of factory `EditFrame` and uses its `facture()` method on `t` to generate a corresponding GUI component class `editFrame`, which is a GUI control for modifying the public member variables of instances of type `t`. The generated class is instantiated by means of the Java reflection API, and the `edit()` method called on that instance with an instance `o` of class `t` as argument.

```
Class editFrameClass =
  new Factory("EditFrame")
    .facture(t);
EditFrame myEditFrame =
  (EditFrame) editFrameClass
    .getConstructor(null)
    .newInstance(null);
myEditFrame.edit(o);
```

## 5 Conclusion

We introduced the Factory language, outlined its syntax and semantics and described how it can be used to perform reflection and partial evaluation. Our idea was to formulate generators as templates. The template approach means that as much as possible of the desired output can be expressed directly. Often, this approach is already used for parametric polymorphism (e.g., in [11]), so it seemed straightforward to integrate into it further support for generative programming. In order to do advanced generation work, we implemented means to perform partial evaluation with the capability to introspect type parameters and to generate new signature elements and code. An important aim was to provide powerful reflection capabilities while still ensuring type-safety, i.e., a new, more general kind of safety that we call generator-type-safety.

We gave some examples in order to demonstrate the power and, above all, usefulness of Factory for real software development. Factory can be useful in the development of a wide range of applications. Like aspect oriented programming [8], it can address crosscutting concerns, i.e., functionality in a software system that is needed at different places, effectively by generating adapted subclasses and can be used dynamically in adaptive systems.

## References

1. Boris Bokowski and Markus Dahm. Poor Man's Genericity for Java. In Serge Demeyer and Jan Bosch, editors, *Object-Oriented Technology, ECOOP'98 Workshop Reader, ECOOP'98 Workshops, Demos, and Posters, Brussels, Belgium, July 20-24, 1998, Proceedings*, volume 1543 of *Lecture Notes in Computer Science*, page 552. Springer, 1998.
2. Luca Cardelli. Type Systems. In *Handbook of Computer Science and Engineering*, chapter 103. CRC Press, 1997.
3. Oracle Corporation. Build Superior Java Applications with Oracle9iAS TopLink. Oracle Whitepaper, September 2002. http://otn.oracle.com/products/ias/toplink/TopLink_WP.pdf.
4. Benjamin C.Pierce. *Types and Programming Languages*. MIT Press, 2002.
5. K. Czarnecki and U. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, 2000.
6. Franois-Nicola Demers and Jacques Malenfant. Reflection in Logic, Functional and Object-oriented Programming: a Short Comparative Study, 1995.
7. Dirk Draheim, Christof Lutteroth, and Gerald Weber. An Analytical Comparison of Generative Programming Technologies. In *Proceedings of the 19. Workshop GI Working Group 2.1.4*. Technical Report at Christian-Albrechts-University of Kiel, November 2002.
8. Kiczales et al. An Overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 18–22. Budapest, Hungary, June 2001.
9. Christof Lutteroth. Factory, 2003. http://www.inf.fu-berlin.de/pj/factory/.
10. Sun Microsystems. Enterprise JavaBeans(TM) Specification 2.1 Proposed Final Draft 2, June 2003. http://java.sun.com/products/ejb/.
11. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, June 1997.
12. Sun Microsystems. *Java 2 SDK, Standard Edition - Documentation*, 2003. http://java.sun.com/j2se/1.4.2/docs/.