Generative Programming for C#

Dirk Draheim, Christof Lutteroth
Institute of Computer Science
Freie Universität Berlin
Takustr.9, 14195 Berlin, Germany
draheim@acm.org lutterot@inf.fu-berlin.de

Gerald Weber
Department of Computer Science
The University of Auckland
38 Princes Street, Auckland 1020, New Zealand
g.weber@cs.auckland.ac.nz

Abstract
This article describes how we extended the C# language by new constructs that provide means for generative programming. Those constructs make it possible to handle problems like the generation of user interfaces and certain crosscutting concerns in a less error prone and elegant way without affecting the overall language integrity.

Keywords: generative programming, genericity, reflection

1 Introduction
In today’s software development new systems are rarely built from scratch but usually with the help of separately developed components. The degree to which a component can be reused depends on how well it can be adjusted to fulfill a desired function. With the increase of component-based development it becomes more and more important to have flexible and powerful means for specifying components and adapting them. The concepts of generative programming [1] target exactly this issue.

With genericity or parametric polymorphism found, for example, in ADA, Haskell or Java, it is possible to program components that are uniformly reusable for many types. However, these generic type parameterization mechanisms are at the same time type abstraction mechanisms: the construction of the type cannot be exploited in the parameterized software component – at most it can be exploited up to a bound, known as bounded parametric polymorphism [6]. Therefore it is very typical that genericity has been successfully exploited in
programming container libraries, e.g., C++ Standard Template Libraries, but not beyond.

Our approach is not about programming containerlike generic components, but truly generative components that address ubiquitous crosscutting concerns, especially mitigating driving forces in software architecture like load, performance, and security. That is, we target a sourcecodebased approach to software generators. Generators are a cornerstone of today’s software engineering. Important examples are generators for transparent data access layers like TopLink and EJB or software distribution mechanisms like the RMI stub generator and the CORBA IDL compiler. In general, a generator gets a proprietary type description as input. It explores the type description and generates source code. For example, a generator for a transparent data access layer gets the description of a semantical data model and generates database tables, queries, access code and appropriate access code capsules. Besides commercially available generators, customized generators are developed from scratch in today’s projects for special purposes. However, the task of programming such generators is not supported adequately by today’s genericity concepts. Support for type introspection and reflection is needed. Using a reflection API misses the point: it is not satisfactory because it is dynamic and not type safe.

Our approach takes another direction. It combines parametric polymorphism and limited reflection capabilities from scratch in a way that is (i) powerful enough to support programming generators, but (ii) terse enough to define a new kind of type safety that can be ensured statically. It can be ensured at generator definition time, i.e., before a generator program is sold, that the generator program will not provoke certain type errors in any of its applications.

In this letter we demonstrate by example how the Factory language can be used to solve generative programming problems in a purely language-based, highly reusable fashion.

2 The Factory Language

Factory [4] introduces a compile-time sublanguage for generation. This makes it possible to write code, parts of which are executed at compile-time, and perform computations that result in the generation of new code. All this happens during compilation and enables the programmer to write and reuse parameterized code segments that generate variants of frequently needed implementation parts, adapted according to some actual parameters.

The Factory generator sublanguage is limited compared to C#. However, this is not a problem because it satisfies all the needs of common generation tasks. The limitations make it possible to check certain safety properties of the generator code statically and consequently to ensure the quality of generators.

Let us understand the Factory language by looking at source code examples. Factory source code is very similar to C# source code in two ways. First, the C# language itself is not changed, but rather extended. This means that any ordinary C# source code is still valid. The syntax for generation is embedded
into the ordinary syntax in a template style, so that the new syntax needs only be used when generation should actually be performed. Secondly, the new language constructs that are introduced by Factory are syntactically and semantically very similar to those of C#, although it is made sure that they can be distinguished from the ones of C# easily. This will become clear when we look at examples.

The first example shows how generic types can be programmed using Factory. Here we define a parameterized class – a class factory – that can be used to generate ordinary classes at compile-time. The Factory language defines several such generators: in addition to class factories we have, for example, factories for generating methods and sets of types. All these kinds of generators accept parameters which are declared similarly to those of ordinary methods. In order to use the parameters of factories and process information in factories we make use of Factory expressions, which form a subset of C# expressions. Such Factory expressions can be enclosed in `@` signs and placed at certain source code positions in order to generate tailored code.

```csharp
public class Stack(Type T)
{
    private Stack s = new Stack();
    public void push(@T@ x) {
        s.push(x);
    }
    public @T@ pop() {
        return (@T@) s.pop();
    }
}
```

In line 1 we declare that the class factory `Stack` has a single type parameter `T`, which can be accessed in the factory body in variable `T`. We call such variables that are used for generation and can therefore only be accessed in generator code generator variables. The main point with generic types is that type variables can be used in place of types. In our case generator variable `T` serves as type variable as it holds a value of metaclass `Type` which represents types in the C# language. In order to use the type given by `T` we insert Factory expressions `@T@` at the places where we want to generate a type. This is done in lines 5, 9 and 10.

Factory expressions can compute values in a more complex manner than in our first example. Besides variable access and literals they can also contain invocations of C# methods and constructors and some basic operators. Also factories can be invoked. Only side effects are prohibited because this could infringe safety. Certain type checks would not be possible any more.

By substituting syntax elements by appropriate Factory expressions, intercession [3] can be performed and we can generate tailored code parts like types, identifiers and values. As we have seen, a Factory expression of type `Type` can
be used to generate a type. A Factory expression of type String can be used to generate an identifier. In order to generate a statically type safe stack myStack of String with the Stack factory, for example, you would write a Factory expression that applies the factory on type String and place that expression, enclosed in @ signs, into the definition of myStack:

@Stack(String)@ myStack = new @Stack(String)@();

The Factory language offers literals for types. String evaluates to a Type object representing the string type. Constructors have the same name as their types, therefore a Factory expression of type Type is used to generate the first argument of the new operator.

In the next example we will see a construct for iterative generation. The example is a method factory for methods ToString that create string representations of objects. The factory has a single parameter of type Type, which represents the type of which the generated method should be member, i.e., the type of the objects for which the method should work. The string representation is created by concatenating the field names and corresponding field values of an object. Such a method is used, for example, when an object is printed on the console, and it can be very useful for debugging purposes.

```java
override public String ToString(Type T)()
{
    String s = "";
    @foreach(F in T.GetFields())
        s += @=F.Name@ + ":	" + this.@F.Name@ + "\n";
    return s;
}
```

In line 4 we use the @foreach control construct, which is similar to the ordinary foreach loop but is executed at compile time. The structure of type T is accessed via the standard C# reflection API, and we iterate over all its fields. Information about a field can be read from the new iterator variable F in the loop body. In line 5 we concatenate a generated string literal, ordinary string literals, and the content of a field whose identifier is generated. A Factory expression enclosed in @= and @ denotes the generation of a literal; in our case @=F.Name@ generates a string literal. The expression this.@F.Name@, on the contrary, generates the identifier of a field.

It is very easy to program factories that generate wrappers. The following example shows a class factory that generates a subclass of a given one. The resulting class implements the functionality of a security proxy. Factory can generally be useful for supporting the implementation of design patterns [5], like the proxy, the observer or the template method pattern.

```java
class SecurityProxy(Type T) : @T@
{
    @foreach(M in T.GetMethods())
        @if(M.IsPublic)
```
override public

@M.ReturnType@ @M.Name@(@M.GetParameters()@) {
    if(/* access permitted */) {
        base.@M.Name@(@M.GetParameters()@);
    } else {
        throw new SecurityException();
    }
}

In line 1 we declare the type parameter \( T \) and specify that the generated class is a subclass of \( T \). In line 3 we iterate over all methods \( M \) of \( T \). We want to protect only those methods that can actually be accessed from outside, therefore the following generation of a wrapper method is conditional and only performed for public methods. We make use of the \@if\ construct for conditional generation. The following part generates a method definition that overrides the respective public method represented by \( M \). In line 6 we generate a return type, an identifier and parameters with appropriate Factory expressions. While a type can be generated with a \Type\ object and an identifier with a \String\ object, a list of parameter declarations can be generated with an array of \ParameterInfo\ objects, as it is provided by method \GetParameters\. As we can see in line 8 where we invoke the original method, such an array can also be used to generate a corresponding list of actual parameters. In line 7 we generate code that checks if the method access is permitted. This part highly depends on the actual application, so we left it out in this example. If permission is given the requested method is called, otherwise an exception is thrown.

Another application for Factory and generative programming in general is the generation of system interfaces, be it a GUI, a database interface, a web interface or an API. The following Factory code sketches out a class factory that generates a GUI form for editing a selection of fields of a given object. Many tasks of interface generation can be done in a similar way. This example can be seen as a merely programming language based instance of model-based user interface development [2] – the model is described in terms of the programming language type system.

class EditForm(Type T, FieldInfo[] V) : Form
{
    foreach(F in V)
        // declare widgets needed to edit field F
    @T@ X;
    @constructor@(@T@ x) {
        this.X = x;
        foreach(F in V) {
            // initialize widgets
            // set event handlers
        }
}
We define a class factory that declares a parameter $T$ for the type of the objects that should be edited. Parameter $V$ holds an array of FieldInfo objects representing the subset of the fields of $T$ that should be editable. First, in lines 3–4, we iteratively generate the widgets that we need for the construction of a suitable GUI. For most fields this could be a Label and a TextBox. In line 6 we declare a member variable that holds the object to edit. In lines 8–14 we generate a constructor. The identifier of the constructor is generated with the keyword constructor. It stores the object to edit and sets up the widgets and their event handlers appropriately. In line 16–17 we generate event handlers that propagate changes in the GUI to corresponding changes in the fields of $X$. In Fig. 1 we see how the input and output of a generation could look like.

3 Conclusion

The integration of a generator sublanguage makes C# a powerful language for the creation of generic components. The kind of generators that can be developed are much more powerful than simply generic types. They are syntactically and semantically based on the host language C# and therefore easy to understand, but enable introspection of types and intercession. Because generators use ordinary C# types they are easily extensible.

With Factory it is possible to facilitate the implementation of object-oriented design patterns, generate various kinds of interfaces and to produce optimized code. While the language constructs are carefully chosen to make these generation tasks possible, they are also designed to render static safety checks possible, so that generation errors can be prevented beforehand.

More information about Factory and a research implementation can be found on the project website, www.factory.formcharts.org.
References


