Jenerator – Generative Programming for Java

Markus Völter[1] and Andreas Gärtner[2]

[1] MATHEMA AG, Germany markus.voelter@mathema.de
[2] Universität Passau, Germany gaertner@fmi.uni-passau.de

Abstract. Generative Programming aims at creating software components, which, after suitable configuration, *generate* systems or other components. This allows to build families of products (or product lines) out of which several concrete systems can be created. Compared to frameworks, this has the advantage that the configuration takes place before runtime, enhancing runtime performance. This paper introduces a tool called Jenerator, an extensible code generator for Java. By using its extension mechanisms, complete high-level, product-line-specific generators can be build, enabling the automated creation of systems on source-code basis.
The second part of the paper presents an application of Jenerator, a tool and framework that can be used to produce components with different layouts out of existing Java classes. Additionally, code for persisting these components can be generated.

1 Introduction to Generative Programming

1.1 Software Product Lines

Reuse has been paramount in software development over the last decades. Many programming techniques, such as modularization, object orientation, etc. were developed primarily to facilitate reuse. These techniques focused on code reuse – during the 90's other reuse techniques emerged, such as design patterns to reuse proven design concepts [GoF94] [BMRSS96] and frameworks to reuse application architectures.

Frameworks are especially interesting, because they provide reuse of code and architecture, and not just reuse of abstract design concepts, as design patterns do. The idea behind frameworks is to create a common base for a family of similar, but not identical applications. A framework contains those parts of an application family that are considered stable for all applications in this family. The varying parts must be programmed – or configured – by the framework user, i.e. the application programmer.

A software product line defines a family of software products, together with the commonalities and the differences between the products. In other industries, a product line based approach is common, and proved to be very successful. The automotive industry uses this approach extensively, the same is true for commercial airplane manufacturers, such as Airbus.

The essence of software product lines is to allow the creation of software from a common family base by combining and configuring components, creating code, etc. This enhances the chance for reuse significantly. One way to realize this concept is the usage of generative programming.

1.2 Generative Programming

Let's begin with a definition of generative programming from Ulrich Eisenecker:

Definition 1. *Generative Programming (GP) is about designing and implementing software modules which can be combined to generate specialized and highly optimized systems fulfilling specific requirements [Eis97].*

The important difference between GP and frameworks (and the configuration of software in general) is expressed by the word *generate*. In the case of frameworks, you combine the components at runtime, using design patterns or other "traditional" technologies. In the case of GP, you create programs or components which *generate* the application *before* it runs. The traditional design techniques are applied to the generating components, not to the application itself.

As there is no support for generic programming in Java available (yet, see [JSR]), static template metaprogramming as described in [CE00] is not possible. Thus, the basis for building generative components in Java is to have a flexible code generation toolset. Based on such a tool, generative components can be easily built. Jenerator is an extensible code generation toolkit for Java.

2 Jenerator Basic Concepts

2.1 Overview

Jenerator is a code generator for the Java programming language. It is itself implemented in Java. On the lowest level, Jenerator provides classes to create classes, methods, members, interfaces, etc. Based on this "foundation", more abstract concepts are implemented, such as macros or aspects, which modify existing classes in specific ways. Applying object-oriented techniques such as inheritance and delegation to Jenerator classes, higher level, domain-specific generators can be implemented and used easily.

2.2 Basic Classes

To introduce Jenerator, let's look at some statements out of "Hello Jenerator", a component that creates a program printing the well-known *Hello, world!* to stdout.

```
...
CClass createdClass = new CClass( "de.hello", "HelloWorld" );
CMethod mainMethod =
```

```
    new CMethod( CVisibility.PUBLIC, CType.VOID, "main" );
mainMethod.addParameter(
    new CParameter( CType.user( "String[]" ), "args" ) );
mainMethod.addToBody( new ClassInstantiation(
    createdClass.getName(), "app", true ) );
CConstructor cons = new CConstructor( CVisibility.PUBLIC );
cons.addToBody(
    new CCode( "System.out.println(\"Hello, world!\");" ) );
createdClass.addConstructor( cons );
createdClass.addMethod( mainMethod );

new CodeGenerator().createCode( createdClass );
...
```

The code that is generated here is the usual hello world program (the generated code is omitted for brevity). This piece of code reveals some of the basics of Jenerator. The basis for code generation is the class *CClass*. An instance of *CClass* plays the role of a container for *CMethods, CConstructors, CMembers*, etc. The purpose of these classes should become clear in the course of this paper.

2.3 Principles for Extension – Building Generative Components

The code presented above is very long for what it does. For example, it is certainly a requirement found quite often during programming, that a class has a default constructor and a main method that does nothing else but creating an instance of the class in which it is located. It is simple to create a subclass of *CClass* which does exactly that.

```
public class SimpleMainInstantiatingClass extends CClass {
    CConstructor cons = null;
    public SimpleMainInstantiatingClass(
      String pName, String cName) {
        super( pName, cName );
        CMainMethod mainMethod = new CMainMethod();
        cons = new CConstructor( CVisibility.PUBLIC );
        mainMethod.addToBody( new ClassInstantiation(
          cName, "instance", cName+"(args)" ) );
        cons.addParameter( "String[] args" );
        addConstructor( cons );
        addMethod( mainMethod );
    }
    public CConstructor getConstructor() {
        return cons;
    }
}
```

This class uses another higher-level generator called *CMainMethod*, because the signature of a Java *main* method is always the same. The constructor can be obtained by the client by calling *getConstructor()* – allowing the client to add custom code to the constructor.

This program is already significantly shorter, because it uses (somewhat) higher-level abstractions. This is the way Jenerator should be used. The following basic extension techniques become obvious:

- **Subclassing:** By subclassing generator classes, higher-level, domain specific generators can be created.
- **Parametrization:** By parameterizing the generator classes, the behaviour of a generator can be easily controlled.
- **Delegation:** Generator classes can use each other, creating more complex results.

In addition to these basics, more advanced techniques exist. These techniques are described below:

- **Macros:** A generator that can be applied to a class, doing a specific modification. It can also act on parts of that class, such as methods.
- **Classgroups:** Often, a certain system consists of several, related classes (e.g. an EJB). Classgroups allow to group these classes and apply macros to all of them.

3 More Advanced Techniques

3.1 Macros

A macro is an entity that modifies a *CClass*. Macros must inherit from the *CodeMacro* class and their behaviour is packaged into the *execute()* operation, according to the *Command* design pattern [GoF94]. As in the *Command* pattern, a concrete macro class can be parameterized by initialization parameters in the constructor.

Example 1 (Properties).

Using JavaBeans, there is a well-known idiom called *Property*. If you have a property called *name* of type *type* then this means you have a private member *type name,* a public operation set*Name(type _name)* and an operation *type* get*Name()*.

The way to add a property to a *CClass* is to create a macro that, in it's *execute* method, adds a *CMember* and get and set methods of type *CMethod* to he class.

3.2 Aspects

According to [AOP], an aspect is "a unit of software modularity that cleanly encapsulates crosscutting concerns". It allows parts of a program that would normally be scattered throughout the code to be localized in one place, the aspect. Typical candidates for aspects are error-checking strategies, design pattern implementations, synchronization policies, resource sharing, distribution concerns, optimization and logging. For more information see [AOP].

One important concept in the context of aspects are join points, i.e. locations where normal and aspect code are woven together. Typical join points include: method invocations, method declarations, object instantiations, thread creations, etc.

In Jenerator, aspects are a special kind of macro, and they allow to attach to basically any kind of join point, as long as the join point is implemented by a distinct class in the Jenerator code tree. For example, if object instantiations are programmed using a *ClassInstantiation* class, then this class can serve as a join point. Each concrete Aspect has to extends the *Aspect* class, overriding some abstract methods. They have the following semantics:

- *introduce()* allows to introduce new operations or members into the classes. It is called only once for each *CClass* to which the aspect is applied. It is comparable to the *execute()* operation in macros.
- *jointPointClass()* returns the class that represents the join point we want to attach the aspect to. An example could be to return *CMethod.class*, so that every method declaraction is affected by the aspect.
- *appliesTo()* has to return *true* if the passed *CodeSnippet* (base class of all entities in a Jenerator source tree) should be affected by the aspect. For example, you can filter the method names; e.g. the aspect would then only apply to methods starting with *set...* . This method is not abstract, it returns *true* by default, so that all instances of the class returned by *joinPointClass()* are affected.
- *apply()* really modifies the passed *CodeSnippet*, for example adding *before* code to a method.

Example 2 (Log Object Creation).
The purpose of the following aspect is to log object creation at the place where the object is created. Here we use *ClassInstantiation.class* as the join point class, because this is, how new objects should be created. The aspect, therefore looks like the following:

```
public class LogObjectCreationAspect extends Aspect {
    public Class joinPointClass() {
        return ClassInstantiation.class;
    }
    public void apply( CodeSnippet cs ) {
        ClassInstantiation inst = (ClassInstantiation)cs;
        CodeContainer container = inst.parent();
```

```
        CMethod m = (CMethod)inst.parent( CMethod.class );
        String log = // log message
        container.addChildBefore( new CCode( log ), cs );
    }
}
```

The aspect shown above also shows how navigation can be performed. Each *CodeSnippet* can be asked for its parent, and also for a parent of a specific type. So we can easily obtain the surrounding *CMethod* or – if we would like – the surrounding *CClass*.

Because the code exists in an internal object form (a *CClass* instance and its children) during code creation, it is easy to attach an aspect to any kind of join point, if this has been enabled by using specific *CodeSnippet* subclasses, such as *ClassInstantiation*. For another example of aspects see chapter 4.5.

3.3 ClassGroups

Often, a specific abstraction cannot be implemented with one class alone, instead it consists of several classes and often additional files. For example, look at an EJB: it consists of the remote interface, the home interface, the bean implementation, and a deployment descriptor (an XML file).

Basically, a class group is a list of name-value pairs, whereas the values are either *CClass*, *CInterface* or further *ClassGroup* instances. A *ClassGroup* provides operations like *putCClass()* or *getCInterface()* to access the elements. To make development easier, macros can also be applied to whole *ClassGroups*; they must then inherit from *GroupMacro* instead of *ClassMacro*.

Example 3 (Entity Bean).
As mentioned above, an entity bean consists of several classes, which are grouped in a so called *EntityBeanClassGroup*, that can be implemented as a subclass of *ClassGroup*. It would now be just as easy to extend the property macro from above, so that it additionally adds the signature of the newly created methods to the remote interface as well.

3.4 Product Lines

As mentioned above, the main goal of generative programming is to enable product-line based software architectures. A product-line is a domain specific family of artifacts. Product-lines can be built on technical level or on a domain-specific level:

- **Technical level:** The family of all possible entity beans (as defined above), which have properties, one (of possibly many) persistence option, and optionally a dirty flag. The configuration space does not say anything about valid property configurations and their dependencies or constraints.

– **Domain-specific level:** The family of *Person* components. The family defines several mandatory properties (such as *name*, first *name*), optional properties (such as *date of birth*) and alternative properties, of which only one is possible. *Person* components could also provide behavioral flexibility, for example the strategy to print the *Person* in a short form could be configurable (*Markus Voelter*, or *Mr. Voelter, Markus*, etc.)

These configurations are usually done by different people (or roles). The domain expert who models a *Person* component does not care about the technical aspects such as persistence; and the programmer who creates a family of entity beans does not care of whether they are used for *Person*s or any other domain abstraction.

However, both levels are conceptually equal. They define a family of systems, whereas a concrete member of the family can be generated by configuration of the product-line components.

Configuration repositories and code creation can be used on both abstraction levels. It is even possible to apply both levels in sequence. First, the domain component is generated from a domain-specific product-line. Properties like *name*, *first name*, and business methods can be generated. As a second step, the class generated in step one can be modified, by making it an entity bean, adding the persistence aspects, etc.

This allows us to model the business classes independent from how they will be used later on, for example as an entity bean in an application server.

4 An Application of Jenerator

Built upon Jenerator's features, a framework was developed that allows the generation of software components with different layouts from existing Java classes. Additionally, code for persisting the data contained in the component can be produced. Thus, the framework can be used to build product lines as described in chapter 3.4. The following sections will introduce the ideas behind the framework and give some insight into the realization.

4.1 Component Technologies

Component technologies such as Enterprise Java Beans (EJB), COM+ or CORBA Components (CCM) aim at simplifying the process of building distributed enterprise applications. They do so mainly by separating concerns: components implement only functional concerns, while the technical concerns such as transactions, persistence, security, threading, or load-balancing are handled by a container, in which the components are deployed to be executed (see [MV01]).

This separation of concerns simplifies reuse, especially of the container, and allows programmers to focus on what they know best - either the functional or the technical parts of an application.

4.2 Component Layouts and Component Persistence

The focus of the framework is the generation of components that represent business entities. Business entities represent concepts (person, invoice, ...) of the problem space. They are mostly defined by the data they contain and may have some simple business logic. The component's data has to be stored in a database.

There is more than one way to model a business entity by means of a component. We will call this the *layout*. The most obvious solution would be to make use of the concept of an entity component, also called an *instance-based* layout. Another solution is to hide access to the component's data by implementing some kind of "manager" for this concept. This is known as a *type-based* layout. Service components that have no persistent state can be used for this approach.

See [HS00] for a discussion of the different layouts and when to use them. Applied to EJB, the type-based style would lead to the generation of a session bean that takes the function of the manager and the instance-based style would be modeled by an entity bean.

An additional aspect of the layout is where to put persistence logic: the code could be directly included in the component's methods, or encapsulated in specialized objects, so-called data access objects [DAO]. The second strategy will lead to another level of indirection, but promises to provide a cleaner separation of concerns and thus a greater flexibility in deployment and a better independence from the resource implementation.

The presented tool takes on both the job of producing a component layout and of generating the persistence logic for a business entity. One feature is the independence of layout and persistence mechanism. Generators for any kind of component layout can be plugged into the framework and combined with another generator that creates the statements necessary to access a particular kind of data store, all without having to write a generator for each of the combinations. Our proposed solution is described in the next sections.

4.3 Parsing a Structure

The core concept of a business entity, and thus a component modeling this entity, is represented by a class structure holding only the classes that are needed for this concept. We use the following class structure as an example: a *Person* class has an association to one *Address* and to a number of contacts. A *Contact* is an abstract class that has two concrete subclasses: *Mail* and *Phone*.

The main concept is modeled by the class at the top of the structure, the *Person* class. We call it the *focus class* of the component. The other classes are *dependent classes*, i.e. they define additional aspects of the main concept, e.g. that a person may have an address. When constructing a component from the given class structure, the focus class will evolve into the main component class and the dependent classes will become so-called business data types, that will be passed by value. The main functionality of the concept is enclosed in the focus class.

Jenerator provides the possibility to parse single classes and transfer them into it's internal object model. As we like to manipulate whole structures, the object model had to be extended to handle references between classes. The main work is done by a class called *StructureReader* that reads and parses the focus class and follows the references to other classes that are defined as members of the focus class.

In our person example this results in reading the associated class *Address* and the interface *Contact*. Additionally, references are set in the object representing the focus class, so that the structure can be traversed later. Another step in the process is to examine classes that subclass one of the classes in the structure, i.e. the classes *Mail* and *Phone*. This is done through the help of reflection. After all this is done, a complete image of the source structure exists in memory and can be freely accessed and modified afterwards.

4.4 Architecture of the Framework

The layout of a component, or the component technology used, is independent from the type of data store that is used to persist the data contained in the component. That means, that it is possible to separate these aspects in the process of generation. All that has to be done is to provide a logical view on a component, that offers the option to "plug-in" any type of specific persistence logic. The same is true for any other aspect (caching, transactions, security, ...).

This is achieved by defining variation points in a component, i.e. methods that have to contain logic for accessing the data store. These include finder methods, create, read update and delete (CRUD) methods and methods for retrieving objects out of the component's environment. This is taken into account by the framework that controls the process of generation.

The main classes of the framework are the *Generator* class that controls the process, and the *LayoutGenerator* and the *PersistenceTypeGenerator* (an instance of the *Strategy* pattern [GoF94]), which are abstract classes that have to be extended by specific generators and take on the job of producing the two aspects of the component.

The generator will first call the *LayoutGenerator*'s methods that produce the layout of the component. Afterwards, the variation points have to be declared by the *LayoutGenerator* and the Generator will use the *PersistenceTypeGenerator* to fill these methods with statements specific for the persistence strategy. The required functionality for the particular generators is explained in the following.

As mentioned above, the dependent classes will become business data types and be passed by value. That means, that those classes have to be serializable. The main Generator class ensures this. Furthermore, all classes will get an additional property for the object identity.

4.5 Generating a Layout and Adding Persistence Logic

The following chapter will introduce the most important methods that a specific *Generators* have to implement. An entity EJB will be used as an example in

the oncoming explanations. A method named *generate()* is called to produce the component classes, e.g. bean class, remote and home interface. The focus class is provided as a parameter and can be used to determine package and name of the class and to generate methods in the component.

For example, a bean class has a number of *ejbCreate* methods that are reflected in the home interface. They can simply be generated by cloning the parameters of the constructors provided in the focus class, i.e. a default constructor in the focus class will lead to the generation of an *ejbCreate* method in the bean class and a *create* method in the home interface, both with no parameters.

After this has been done, the Generator will call all of the methods that declare the variation points, i.e. the methods that will have to be filled with persistence logic. These must also be implemented by the specific *LayoutGenerator*. The code fragments are then produced by the *PersistenceTypeGenerator*. The diagram in fig. 1 illustrates the sequence of the method calls.

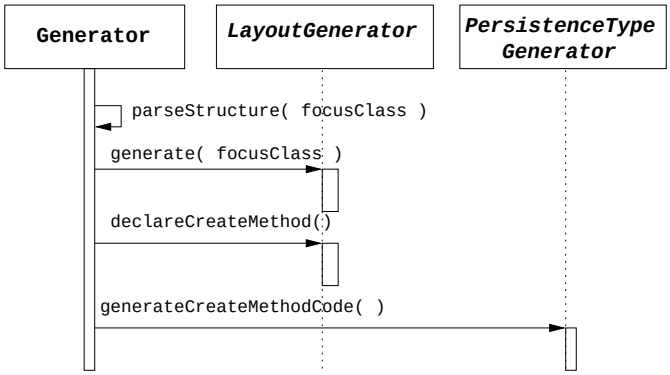

Fig. 1.

The *PersistenceTypeGenerator* includes callback methods for each component method that has to be filled with persistence logic. The actual statements that insert, read, update, delete or find an object have to be produced by the implementation of these callback methods in the specific generator.

Crosscutting concerns (aspects) that are independent from the chosen persistence strategy can later be added by executing aspect macros as described in chapter 3.2. A typical example is the storage optimization using a “dirty flag:

Example 4 (Storage Optimization in Entity Beans).

In entity beans, the method *ejbStore()* is called by the container whenever the bean has to persist its internal state to the database. It is useful, to track changes to a bean by using a *dirty* flag. This optimization can be executed by an aspect.

5 Future Work

This paper describes an extensible architecture for Java source code generators. The next step is to make configuration repositories usable for the users of a specific product-line. Feature diagrams are a proven way to visualize the configuration space of a family of products.

A generator tool based on Jenerator's features was introduced that provides the means to automatically produce software components with additional persistence logic. The architecture was developed to achieve an easy extensibility. Generators for new component layouts or component technologies can be added and combined with existing or newly created generators for persistence logic. The sample implementation features layout generators for entity EJBs and manager-like components and generators for relational database persistence (via JDBC) and JDO persistence.

The next step in our work will be to enhance the component generation framework with additional features, because components typically contain a lot of code that can be generated – once it's requirements are defined. We plan to integrate the component generator with the MATHEMA Component Configuration Framework (MCCF). The MCCF is a toolset that provides a GUI based feature model editor to define potential features of components. Currently, it only supports component-internal aspects (such as logging, strategy objects, etc.). The next step will be to use it to define a configuration repository for complete components. The tool can be used to define the configuration space as well as to select and define possible component configurations. Typical configuration spaces will include technical concerns such as persistence, target container, and deployment descriptor options as well as functional concerns specific to the component. Before generation starts, the tool checks the consistency and correctness of the configuration based on user-defined constraints. Actual code generation will be done by the previously described component generator.

6 Related Work

COMPOST [COMPOST] is a software composition and transformation system developed by the universities Karlsruhe and Linkping. It uses compile-time metaprogramming to modify, assemble, and adapt Java source code. While Jenerator uses code generation, COMPOST transforms existing source based on specific rules. It includes a meta model of the underlying source code on which it executes its transformations.

The idea of generic components has been addressed in several publications, e.g. [CE99], [Be00]. They propose a tailoring of components by providing means to configure them through generic parameters or code generation.

Introducing persistence services into Java is an even broader field of research. Most solutions target the storage in relational databases. Among these are [Blend], [UFO-RDB] and [JRF].

More recent approaches are to provide a unified access layer for all kinds of storage mechanisms [Castor] and even introduce this feature in component technologies [JDO01], [ABMP].

References

[ABMP] Compoze Software, Inc., *Automatic Bean-Managed Persitence* http://www.compoze.com, http://www.theserverside.com/resources/ABMP.jsp

[AOP] XEROX Corporation, *Aspect-Oriented Programming Homepage*, http://www.parc.xerox.com/aop

[Be00] Becker M., *Generic Components: a symbiosis of paradigms* 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE'00), Erfurt, 2000. http://www.uni-kl.de/AG-Nehmer/Projekte/GeneSys/Papers/GCSE00.pdf

[Blend] SUN Microsystems, Inc., *Java Blend Homepage* http://www.sun.com/software/javablend

[BMRSS96] Buschmann, Meunier, Rohnert, Somerlad, Stal, *Pattern-Oriented Software Architecture*, John Wiley & Sons Inc., New York 1996.

[Castor] The Exolab Group, *Castor Homepage* http://castor.exolab.org

[CEGVV98] Czarnecki, Eisenecker, Glück, Vandevoorde , Veldhuizen, *Generative Programming and Active Libraries*, In O.Nierstrasz, editor, Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE 99, Toulouse, France, Sep. 99), Springer-Verlag, 1999. http://www.extreme.indiana.edu/ tveldhui/papers/dagstuhl1998/

[CE99] Czarnecki K., Eisenecker U., *Components and Generative programming*, Software Engineering Notes, vol. 24, no. 6, 1999.

[CE00] Czarnecki, Eisenecker, *Generative Programming,* Addison-Wesley, 2000.

[COMPOST] *Compost, the Software composition system Homepage*, http://i44www.info.uni-karlsruhe.de/ compost/

[DAO] SUN Microsystems, Inc., *J2EE Blueprints design pattern – Data Access Object* http://java.sun.com/j2ee/blueprints/design_patterns/dao

[Eis97] U. Eisenecker, *Generative Programming (GP) with C++*, in Proceedings of Modular Programming Languages (JMLC'97, Linz, Austria, March 1997), H. Mössenböck, (Ed.), Springer-Verlag, Heidelberg, 1997.

[GoF94] Gamma, Helm, Johnson, Vlissides; *Design Patterns*, Addison-Wesley, 1994.

[HS00] Herzum, Sims, *Business Component Factory*, John Wiley & Sons Inc., New York, 2000.

[JDO01] Specification lead: Craig Russell: *Java TM Data Objects* JSR000012 Version 0.8 Public Review Draft, Java Data Objects Expert Group, SUN Microsystems Inc, http://access1.sun.com/jdo

[JRF] IS.com, Inc., *JRelationalFramework*, http://jrf.sourceforge.net

[JSR] Sun Microsystems, Inc., *Add Generic Types to the JavaTM Programming Language*, JSR # 000014, http://java.sun.com/aboutJava/communityprocess/jsr/jsr_014_gener.html

[MV01] Markus Voelter *Server-Side Components - A pattern language*, http://www.voelter.de/cpl

[UFO-RDB] Priese C.P., *Architecture of a reusable and extensible DataBase-Wrapper with rule-set based object-relational schemes*, L'Objet Scientific Journal, 6(3), Hermes Science Publishing, Paris and Oxford, 2000.