

# Jenerator – Generative Programming for Java

*Position Paper for the OOPSLA 2001 workshop on  
Generative Programming*

(c) 2000 Markus Völter, MATHEMA AG, Germany  
markus.voelter@mathema.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.

**Keywords:** generative programming, code generation, frameworks, Java, product lines.

## 1 Jenerator basic concepts

### 1.1 Overview

Jenerator is a code generator for the Java<sup>®</sup> 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, very higher level, domain-specific generators can be implemented and used easily.

### 1.2 Basic classes

To introduce Jenerator, let’s look at “Hello Jenerator”, a component that creates a program printing the well-known *Hello world* to stdout.

```
package de.mathema.jenerator.paper;
// imports...
public class HelloJenerator {
    public HelloJenerator() {
        CClass createdClass = new CClass( "de.mathema.jenerator.paper", "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 );
    }
    // main method following here
}
```

The code which is generated here is simply the usual hello world program (the generated code is omitted here for brevity). This piece of code reveals some of the basics of Jenerator. The basis for code generation is the *CClass* class. 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. The *io* subpackage contains utility classes for outputting the created code into a file, or stdout, as seen above. For more information on the *io* classes, see appendix.

### 1.3 Principles for extension – building generative components

The above code is very long for the kind of thing 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 *CClass*-subclass which does exactly that.

```
package de.mathema.jenerator.util;
// imports...
public class SimpleMainInstantiatingClass extends CClass {
    CConstructor cons = null;
    public SimpleMainInstantiatingClass( String pName, String cName,
                                       boolean passArgs ) {
        super( pName, cName );
        CMainMethod mainMethod = new CMainMethod();
        cons = new CConstructor( CVisibility.PUBLIC );
        if ( !passArgs ) {
            mainMethod.addToBody( new ClassInstantiation( cName, "instance",
                                                         true ) );
        } else {
            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 *SimpleMainInstantiatingClass* takes one more parameter than the normal *CClass*, it determines whether the *args*-array should be passed to the newly created instance. The constructor can be obtained by the client by calling *getConstructor()* – allowing the client to add custom code to the constructor, as demonstrated in the next version of *HelloJenerator*:

```
package de.mathema.jenerator.paper;
// imports
public class HelloJenerator2 {
    public HelloJenerator2() {
        SimpleMainInstantiatingClass createdClass = new
            SimpleMainInstantiatingClass( "de.mathema.jenerator.paper",
                                         "HelloWorld", true );
        createdClass.getConstructor().addToBody(
            new CCode( "System.out.println(\"Hello World!\");" ) );

        new CodeGenerator().createCode( createdClass );
    }
    // main method
}
```

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 parametrizing 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:

- ✂ **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.
- ✂ **Configuration Repositories:** Of course, it is possible to store configuration options in a central repository, and let several generators or macros use this configuration to adapt their own actions.

These more advanced techniques are described below, each with an extensive example.

## 2 More advanced techniques

### 2.1 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 parametrized by initialization parameters in the constructor.

#### Example: 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 *setName(type\_name)* and an operation *type getName()*. For example, storing the *id* of an object as a property will result in the following piece of code:

```
class SomeClass {
    private String id;
    public void setId( String _id ) { id = _id; }
    public String getId() { return id; }
}
```

The way to add a property to a *CClass* is to create a macro that looks like the following:

```
package de.mathema.jenerator.util;
// imports...
public class Property extends CodeMacro {
    protected String name;
    protected CType type;

    public Property( String nm, CType tp ) {
        name = nm;
        type = tp;
    }

    private String upc( String s ) {
        return s.substring( 0,1 ).toUpperCase()+s.substring( 1 );
    }

    public void execute( CClass c ) throws Exception {
        c.addMember( new CMember( CVisibility.PRIVATE, type, name, type.defaultInit() ) );
        CMethod getter = new CMethod( CVisibility.PUBLIC, type, "get"+upc(name) );
        getter.addChild( new CCode( "return "+name+";" ) );
        c.addMethod( getter );
        CMethod setter = new CMethod( CVisibility.PUBLIC, CType.VOID, "set"+upc(name) );
        setter.addParameter( new CParameter( type, "_" + name ) );
        setter.addChild( new CCode( name+" = _"+name+";" ) );
        c.addMethod( setter );
    }
}
```

Using this macro is very simple. Just call *execute* with the class you want to modify as a parameter:

```
CClass createdClass = new CClass( "de.mathema.jenerator.paper", "PropertyTestGen" );
new Property( "name", CType.STRING ).execute( createdClass );
```

## 2.2 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. I don’t want to go into details about aspect-oriented programming, for more information see [AOP].

One important concept in the context of aspects are join points. Aspects represent cross-cutting concerns of a program. To be used in a program, aspects have to be “integrated” with the normal program. The locations in code where aspects are connected to the ordinary programs are called join points. In theory, every location in the code can be a join point. However, in practice, specific join points have emerged to be practical: method invocations, method declarations, object instantiations, thread creations, etc.

In Jenerator, aspects are a specific form of macros, and they allow to attach to basically any kind of join point, as long as the join point is implemented through a distinct class in the Jenerator code tree. For example, if object instantiations are programmed using a *ObjectInstantiation* class, then this class can serve as a join point. Each concrete Aspect has to extend 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.

*joinPointClass()* 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 declaration is affected by the aspect. More examples follow below.

*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: 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:

```
package de.mathema.jenerator.paper;
// imports...
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 );
        CClass c = (CClass)inst.parent( CClass.class );
        String log = "System.out.println( new java.util.Date()+\""+
            ": creating instance of "+inst.getClassName()+" in method "+
            m.getName()+" in class "+c+"\");";
        container.addChildBefore( new CCode( log ), cs );
    }
}
```

The above aspect 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 the surrounding *CClass*.

If we would only want to log the creation of specific types (those whose class name ends with *AClass*), we would have to change the *appliesTo()* operation:

```
public boolean appliesTo( CodeSnippet cs ) {
    ClassInstantiation inst = (ClassInstantiation)cs;
    return inst.getClassName().endsWith( "AClass" );
}
```

```
}

```

Because during code creation, the code exists in an internal object form (a *CClass* instance and its children) 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*.

### Example: 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 into the database. For example, this happens whenever the bean is about to be evicted from memory. There is some space for optimization here, because if the bean has not been changed, then there is no need to update the database. It is therefore useful, to track changes to a bean by using some kind of *dirty* flag. If we assume, that changes to a bean only happen in *set...* or *update...* operations, this optimization can be performed generically by an aspect.

```
package de.mathema.jenerator.ejb;
// imports...
public class DirtyFlagAspect extends Aspect {
    public void introduce( CClass target ) {
        target.addMember( new CMember( CVisibility.PRIVATE,
                                       CType.BOOLEAN, "dirty", CType.BOOLEAN.defaultInit() ) );
    }

    public Class joinPointClass() {
        return CMethod.class;
    }

    public void apply( CodeSnippet cs ) {
        CMethod m = (CMethod)cs;
        if ( m.getName().equals( "setEntityContext" ) ) return;
        if ( m.getName().equals( "ejbStore" ) ) {
            m.addToBodyBegin( new CCode( "dirty = false;" ) );
            m.addToBodyBegin( new CCode( "if ( !dirty ) return;" ) );
        } else if ( m.getName().equals( "ejbLoad" ) ) {
            m.addToBody( new CCode( "dirty = false;" ) );
        } else if ( m.getName().startsWith("set") || m.getName().startsWith("update") ) {
            m.addToBodyBegin( new CCode( "dirty = true;" ) );
        }
    }
}
```

The aspect introduces a new boolean member called *dirty* into the affected class. Then it takes all *CMethods* as join points and does the following things:

- ✂ If the method is *ejbStore()*, check the *dirty* flag and return if it is *false* (i.e. bean is unchanged). Then the flag is set to *false*, because after *ejbStore()*, the bean is not dirty anymore.
- ✂ In *ejbLoad()*, set the flag to *false*, because after loading, the bean is never dirty.
- ✂ All operations that start with *set...* or *update...* set the flag to *true*.

The aspect can be made a bit more flexible, by allowing the client to pass a list of method names to the constructor, which do not start with *set/update*, but still modify the bean. An example of using this aspect will be presented in the next section.

## 2.3 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: Entity Bean

As mentioned above, an Entity Bean consists of several classes, which are grouped in a so called *EntityBeanClassGroup*, a subclass of *ClassGroup*. Let's start with how this is used. The following

piece of code generates the classes needed for an Entity Bean called *Person*, with the attributes *name*, *firstName* and *dateOfBirth*. For brevity, the persistence aspect is skipped.

```
package de.mathema.jenerator.ejb.test;
// imports...
public class PersonTest {
    public PersonTest() {
        try {
            EntityBeanClassGroup g = new EntityBeanClassGroup( "de.mathema.ejbTest", "Person" );
            g.addProperty( "String", "name" );
            g.addProperty( "String", "firstName" );
            g.addProperty( "java.util.Date", "dateOfBirth" );
            // output classes
        } catch ( Exception ex ) {
            ex.printStackTrace();
        }
    }
}
```

The *EntityBeanGroup* extends a more general *ClassGroup*, the *EJBClassGroup*. It applies the macro *MakeEntityBean* to itself, which defines the standard operations required for Entity Beans, such as *setEntityContext()*, *ejbLoad()*, etc.

```
package de.mathema.jenerator.ejb;
// imports...
public class EntityBeanClassGroup extends EJBClassGroup {
    public EntityBeanClassGroup( String pName, String remoteName ) throws Exception {
        super( pName, remoteName );
        new MakeEntityBean().execute( this );
    }

    public void addProperty( String _type, String _name ) throws Exception {
        new AddEBProperty( _type, _name ).execute( this );
    }
}
```

The *addProperty()* operation is implemented in terms of another macro, the *AddEBProperty* macro, which adds setter/getter declarations to the remote interface, and adds a private member and setter/getter operations to the bean class.

## 2.4 Configuration Repositories

Usually, an abstraction, such as an Entity Bean, has several configuration options, and has to enforce constraints about which configurations are valid. A configuration repository is a data structure, that defines which configuration options are possible, and which combinations are valid. Code generation is controlled by the configuration repository.

```
package de.mathema.jenerator;
public abstract class ConfigRepository {
    private ClassGroup result = null;
    public abstract boolean configurationValid();
    protected abstract void execute() throws Exception;

    public void setResult( ClassGroup res ) {
        result = res;
    }
    public final void run() throws Exception {
        if ( !configurationValid() ) throw ( new InvalidConfigException() );
        execute();
    }
    public ClassGroup getResult() throws Exception {
        if ( result == null ) run();
        return result;
    }
}
```

*configurationValid()* determines whether the configuration is ok, *execute()* implements the creation of the resulting *ClassGroup*, which can be accessed using *getResult()*.

### Example: Entity Bean

We specify that a configuration for an Entity Bean consists of

~~several~~ *Properties*

- ✂ one *PersistenceOption*
- ✂ optionally a *DirtyFlag*
- ✂ and of course a package and a name

Building on the example from the *ClassGroups* section, a configuration repository for an Entity Bean could be defined by the following class:

```
package de.mathema.jenerator.ejb;
// imports...
public class EntityBeanConfig extends ConfigRepository {
    private List properties = new ArrayList();
    private boolean useDirty = false;
    private PersistenceOption pers = null;
    private String name = null;
    private String packageName = null;

    public boolean configurationValid() {
        return (properties.size() > 0) && (pers != null) &&
            ( name != null ) && ( packageName != null );
    }

    // setters for name, package, persistence option and dirty flag

    public void addProperty( String _type, String _name ) {
        properties.add( new AddEBProperty( _type, _name ) );
    }

    protected void execute() throws Exception {
        EntityBeanClassGroup result = new EntityBeanClassGroup( packageName, name );
        Iterator it = properties.iterator();
        while ( it.hasNext() ) {
            GroupMacro gm = (GroupMacro)it.next();
            gm.execute( result );
        }
        if ( useDirty ) new DirtyFlagAspect().execute( result.getCClass( "bean" ) );
        pers.execute( result );
        setResult( result );
    }
}
```

So, the creation of an Entity Bean has become even more simple. The code now looks like the following:

```
package de.mathema.jenerator.ejb.test;
// imports...
public class PersonTest {
    public PersonTest() {
        try {
            EntityBeanConfig config = new EntityBeanConfig();
            config.setName( "Person" );
            config.setPackage( "de.mathema.jenerator.ejb.generated" );
            config.useDirtyFlag();
            config.addProperty( "String", "name" );
            config.addProperty( "String", "firstName" );
            config.addProperty( "java.util.Date", "dateOfBirth" );
            config.setPersistenceOption( new XMLPersistenceOption() );
            EJBClassGroup gc = (EJBClassGroup)config.getResult();
            // output bean...
        } catch ( Exception ex ) {
            ex.printStackTrace();
        }
    }
}
```

Even a programmer who does not exactly know the dependencies among the attributes of the *EntityBeanConfig*, can use this generator because before generation, the validity of the configuration is checked.

## 2.5 Mixing normal and generative Programming

Jenerator can be used to mix ordinary and generative programming in one program. To do this, Jenerator can generate, compile and load classes more or less on the fly. This feature is not yet

completely integrate with the Java programming language, as no specific classloader is provided, yet. However, the feature can be used quite effectively.

In the following piece of code, generator is used to generate a class which serves as a model for an editor for the FAF2 application framework. Such an editor model is relatively straight forward, writing it is quite tedious. This is an ideal use for code generation.

```
package de.mathema.jenerator.test;
// imports...
public class FAFUIMain {

    public FAFUIMain() {
        // initialize FAF2 framework
        // ...
        // instantiate a Person
        Person person = new Person();
        // create the editor - use jenerator
        FAFEMBase b = (FAFEMBase)new Loader().instantiate( personEditorClass() );
        // initialize editor with object to be edited
        b.setObject( person );
        // show editor (framework specific)
        FAF.currentFrame().newPane( new EditorRightPane( "", b ) );
    }

    private FAFEMClass personEditorClass() {
        FAFEMClass c = new FAFEMClass( "de.mathema.jenerator.test.Person",
            new String[]{"name /String /rw/Name /TextFieldWidget(30)",
                "vorname/String /rw/Vorname /TextFieldWidget(20)",
                "sex /String /rw/Geschlecht/ComboBoxWidget()|m:w",
                "member /Boolean/rw/Member? /CheckBoxWidget(\"J\", \"N\")"},
            "Person editor" );

        return c;
    }
}
```

The *personEditorClass()* operation creates an instance of the *CClass*-subclass *FAFEMClass*. It is initialized with a specification of the attributes the editor should display. The implementation of *FAFEMClass* makes some assumptions about the class to be edited (here: *Person*):

- ✂ For each property (*name*, *vorname*, etc.) there must be a setter/getter operation pair.
- ✂ The generated editor model class will have the name *Person\_EM*, the implementation always append *\_EM* to the class being edited.
- ✂ The package for the editor class is the package of the class to be edited, appended with *.generated*. Thus, the *Person\_EM* class will be in *de.mathema.jenerator.test.generated*.

The complete implementation of *FAFEMClass* is beyond the scope of this paper – it is very FAF2-specific. However, take a look at the bold line in the constructor. A *Loader* is used to create the class (see Appendix for more information on the *Loader*). Its *instantiate()* operation has the following semantics (error handling omitted for brevity):

- ✂ Try to dynamically load the class with the specified name (here: *de.mathema.jenerator.test.Person\_EM*).
- ✂ If class cannot be loaded, generate it into a source directory. Compile it, and write the class files to a directory from the applications classpath.
- ✂ Try to load it again. If this is not possible, throw *Exception*.
- ✂ Instantiate an object of the loaded class and return it.

The instance can now be used just as any other instance. Of course, it is possible to modify the specification for *FAFEMClass* before it is used for generation. So, the created editor model can be adapted before it is created. To enforce a regeneration of the source code, the loader has to be instantiated with *true* as a parameter:

```
FAFEMBase b = (FAFEMBase)new Loader(true).instantiate( personEditorClass() );
```

## 2.6 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 guy who creates a family of Entity Beans does not care of whether they are used for *Persons* 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.

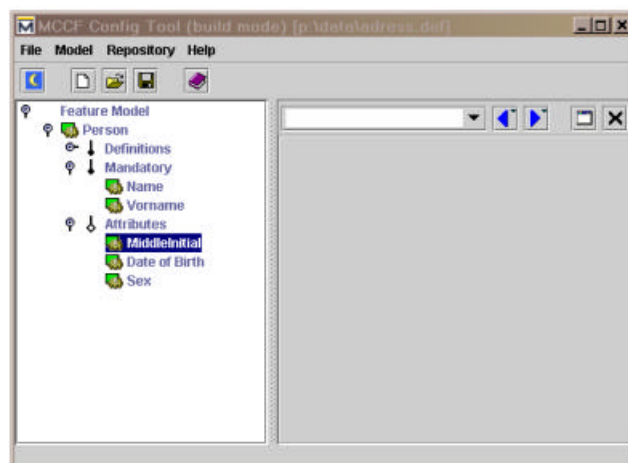
✂ As a second step, the class generated in step one can be modified, by making it an entity bean, adding the persistence aspects, etc.

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.

### 3 Future work

This paper describes an extensible architecture for source Java 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.

The following illustration shows a GUI feature modelling tool. It allows to build feature models, and to configure those for a specific component.



The next step in our work will be to integrate code generation and especially configuration repositories with this feature modelling tool to simplify the use of the code generators.

## 4 Literature and Online Resources

EIS97	U. Eisenecker. <i>Generative Programming (GP) with C++</i> . In Proceedings of Modular Programming Languages (JMLC'97, Linz, Austria, March 1997), H. Mössenböck, (Ed.), Springer-Verlag, Heidelberg 1997,
CEGVV98	Czarnecki, Eisenecker, Glück, Vandevoorde, Veldhuizen, <i>Generative Programming and Active Libraries</i> , <a href="http://www.extreme.indiana.edu/~tveldhui/papers/dagstuhl1998/">http://www.extreme.indiana.edu/~tveldhui/papers/dagstuhl1998/</a>
CE2000	Czarnecki, Eisenecker, <i>Generative Programming</i> , Addison-Wesley, 2000
COMPOST	<i>Compost, the Software composition system</i> , <a href="http://i44www.info.uni-karlsruhe.de/~compost/">http://i44www.info.uni-karlsruhe.de/~compost/</a>
JSR	Sun Microsystems, <i>Add Generic Types to the Java™ Programming Language</i> , JSR #000014, <a href="http://java.sun.com/aboutJava/communityprocess/jsr/jsr_014_gener.html">http://java.sun.com/aboutJava/communityprocess/jsr/jsr_014_gener.html</a>
PIZ	<i>Pizza</i> , <a href="http://www.ipd.ira.uka.de/~pizza/">http://www.ipd.ira.uka.de/~pizza/</a>
GJ	<i>GJ, A Generic Java Language Extension</i> , <a href="http://www.ipd.ira.uka.de/~pizza/gj/">http://www.ipd.ira.uka.de/~pizza/gj/</a>
GoF94	Gamma, Helm, Johnson, Vlissides; <i>Design Patterns</i> ; Addison-Wesley 1994