A Catalog of Patterns for

Program Generation

Markus Voelter
voelter - ingenieurbüro für softwaretechnologie
Ziegelaecker 11, 89520 Heidenheim, Germany
voelter@acm.org, www.voelter.de

Generation of source code is becoming a more and more important tool for software engineering, especially in the context of model driven development and OMG's Model Driven Architecture (MDA). While it is possible in theory to „generate code" just using printf, this approach is insufficient in all but the most trivial cases. However, there are several alternatives. This paper introduces a set of patterns that describe the most commonly used source code generation techniques. That can be used for model transformations, model-to-source generation or source code transformation. It also contains large amount of examples that show how different tools realize the patterns.

Introduction

Program generation is concerned with approaches, techniques and tools for generating program source code which is subsequently compiled or interpreted. In current software development practice, this approach becomes more and more important for reasons outlined below.

Target Audience
This paper should be read by developers and tool developers. Developers will understand the different means available to generate code, while tool developers will understand how their to-be-developed tool could be structured.

Why to use code generation?
There are several reasons why you would want to generate code (source or binary):

- The desired system (or a member of a family) has to have a large degree of variability and you cannot afford to use traditional, generic OO code for performance reasons.

- Another reason can be the minimization of the size of the program executable because the target device only has a limited amount of memory.

- Especially in the embedded world, a generic approach might not be usable because you want to statically analyze the code with regards to

resource consumption, concurrency, etc., Generated code can be much more easily analyzed with regards to these issues compared to generic frameworks. Avoidance of dynamic memory allocation at runtime, while still being flexible with regards to the objects used can also be a reason why to use a code-generation based approach.

- You want to develop application logic at a higher level of abstraction than the level provided by the programming language use, for example because you want domain experts to "program". Code generation can then be used to „reduce" the abstraction level of a specification (or domain-specific language) by generating an „implementation." A typical example is generating source code from (UML) models.

- You cannot express specific things with a programming language because its type system or language features does not allow it (e.g. you want to use the notion of a *class* in a language that does not support classes natively).

- You have a highly modular system, and you only want to include specific „aspects" in your image. Normal modularization (i.e. the modularization with the means provided by the programming language such as classes or functions) is not possible because of crosscutting concerns. Generator-based AOP tools such as AspectJ are typical examples.

- You want to detect design or implementation errors early, i.e. not at runtime, but during the generation of the program code and/or the subsequent compilation.

Alternatives to code generation include the use of powerful libraries or frameworks, reflective facilities as well as interpreters. However, most of these alternatives require more sophisticated runtime systems or have a larger footprint, which makes them unusable in several scenarios. It is not always trivial to decide if code generation should be used, and how. The effort of writing the generator can be quite significant, and if used wrong, benefits like improved performance or reduced code size can easily be turned to the opposite.

Software System Families

Code generation is especially important in the context of product line engineering[1], because the work of developing domain-specific generators usually does not pay-off for one-off products. Product Line Engineering aims at developing families of software systems in a specific domain. A software system family is defined as follows:

> We consider a set of programs to constitute a family whenever it is worthwhile to study programs from the set by first study-

---

1. Whenever we talk about code generation, especially in the context of product line engineering, the generation of artifacts is not limited to program code: manuals, configuration files, maintenance plans can also be generated. In the context of this paper, we limit our scope to code generation, however.

ing the common properties of the set and then determining the special properties of the individual family members. [Parnas, 1976]

Software System Family Examples The biggest payoff of using code generation is typically achieved if the generator can be reused in the context of a software system family. While Parnas' definition above is correct, it is also very general. Let's therefore look at some examples of software system families:

- A set of projects in the same domain can constitute a software system family (banking, telecom switching, automotive diagnosis). You might be able to generate recurring business logic from models.

- A set of artifacts based on the same infrastructure (such as EJB) in one project can also be a family. Here, you might be able to generate all the infrastructure-specific code around manually implemented business logic.

- It can also be considered a family if you have some specific business logic that you want to run on different platforms. You might be able to generate platform-specific implementation code from the models (this is the focus of MDA)

- And finally, a set of artifacts based on the same modelling paradigm, such as state charts can constitute family. You might be able to generate the complete implementation based on the model and its predefined mapping to lower-level implementations.

Generative Programming Today, product line engineering is typically seen in the greater context of generative programming. Let's look at another definition:

Generative Programming is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge. [Carnecki & Eisenecker, 2000]

This definition mentions that „a highly customized and optimized [...] product can be automatically manufactured". This automatic manufacturing of software products can be achieved with different means: object-orientation and polymorphism, frameworks, reflection, and code generation[1]. The decision as to which mechanism to use depends on the time when you are able to decide about a specific feature. If you can decide before runtime, you can use code generation to create a more efficient product, compared to using runtime OO features.

---

1. I is important to note that generative programming does not imply, or even require, source code generation; this is a common misconception. Often, however, code generation is the only means to achieve the goals mentioned in the definition.

If code generation is used, it can happen at different times in the development process, it can happen at different levels of granularity, can be based on different kinds of specifications what should be generated, and there can be several reasons, why generation is used - see beginning of the introduction.

When, where and what to generate

Model and Code defined To understand the following discussions, we have to define what we understand by a *model* and what we consider *code*. We consider a model to be an artifact that has to be transformed to something more concrete to be executed. Code, on the other side is something that can be compiled and executed directly. This definition is important, since it says that UML models are not considered code and a programming language interface is not considered a model. In the context of multi-step generations, there are many model-to-model transformations and one final source code generation step. The generated code is then compiled and executed. Another thing to mention is that we do not require models to have a grahical notation. A model is considered equivalent to a specification in a textual domain-specific language (DSL). The grammar of the DSL thus resembles the metamodel of the model.

Note that this definition of a model and code is somewhat arbitrary. The compilation of Java byte code from Java source can also be seen as code generation, source code can be considered a (rather low-level) model, and it is possible to directly interpret UML diagrams (see [Riehle, 2001]). However, in current mainstream practice, there still is a distinction between models and code in the way we defined it above.

The following illustration shows where code generation is applicable in the application development lifecycle, the following paragraphs give some additional explanation.
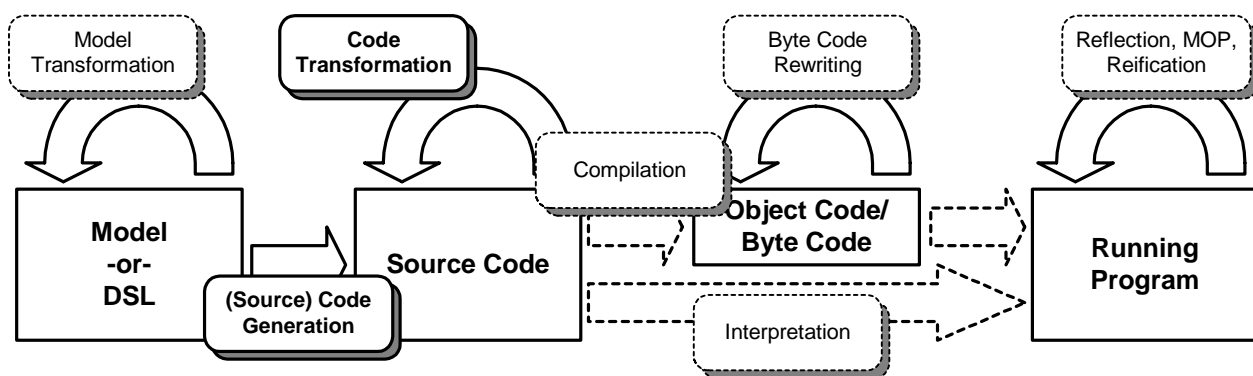
Model Transformation *Model Transformation* generates new models from available models. Typically, these generated models are specialized (i.e. more concrete) with regards to some property, for example, the implementation platform used as is the case in MDA's PIM to PSM transformations. Strictly speaking, this is not code generation because the result is not code per se - it is another mod-

el. However, it uses some of the same techniques as real code generation, this is why this is included.

Source Code Generation

Source Code Generation describes the generation of executable code. This usually involves some kind of de-abstraction or concretization of the model. The generated source usually requires compilation before it can be executed. However, sometimes the generator creates byte code or machine code directly.

Source Code Transformation

Source Code Transformation denotes the creation from code based on other code. This is typically used when different code artifacts are somehow merged or modified.

Byte Code Rewriting

Byte Code Rewriting is a technique that has been introduced in the context of byte-code interpreters (or virtual machines), and has gained recent popularity in the context of platforms such as Java and .NET. Here, the code that has been created by compiling source is directly created or modified.

Reflection

Last but not least, *Reflection* (and the related techniques of intercession, reification, introspection) explains how a program can modify itself while running. Examples for this can be found in Smalltalk or Lisp/CLOS. This technique is not considered in this paper.

The following section contains the basic patterns for code generation whereas later sections cover examples. The patterns denote specific techniques that might be applicable to one or more of the phases in the above diagrams.

Integrated (Meta-) Modeling Environments

In addition to the code generation techniques mentioned above, there are several integrated metamodelling environments available (such as GME [ISIS, web] or MetaEdit+ [Metacase, web]). These tools provide generic, integrated environments for building, validating and managing models based on custom metamodels. They can also be used to generate source code from

UML Modeling Tools

these models. However, these tools are beyond the scope of this paper. The same is true for UML tools that can generate code such as Rose, XDE or Together/J. While all of these tools can generate source code (some even in a customizable manner) they all use some of the techniques mentioned below under the hood.

The Patterns

In this section, we list seven rather basic patterns for code generation. Note that we do not provide extensive examples in the pattern text, subsequent sections describe examples for the patterns, the concrete technological realizations. Let us first get a quick overview over the patterns that follow.

TEMPLATES + FILTERING describes the simplest way of generating code. Code is generated by applying templates to textual model specifications (often XML/XMI), typically after filtering some parts of the specification. The code to be generated is embedded in the templates.

TEMPLATES + METAMODEL is an extension of the TEMPLATES + FILTERING pattern. Instead of applying patterns directly to the model, we first instantiate a metamodel from the specification. The templates are the specified in terms of the metamodel. The metamodel can be extended to include domain or architecture specific aspects

FRAME PROCESSING describes a way of generating code by means of so-called frames. Frames can be seen as programs (functions) that generate code as the result of their evaluation. Frames can be parametrized by number and string literals as well as other frame instances.

API-BASED GENERATORS provide an API against which code-generating programs are written. This API is typically based on the metamodel/syntax of the target language.

INLINE CODE GENERATION describes a technique where code generation is done implicitly during interpretation or compilation of a regular, non-generated program, or by means of a precompiler. This process typically modifies the program that is then subsequently compiled or interpreted.

CODE ATTRIBUTES describe a means by which normal, non-generated program code contains annotations, or attributes, that specify things that are not contained in the code per se. Based on these attributes, additional code can be generated.

Finally, CODE WEAVING is about combining, or weaving, different parts of program text together. These different parts typically specify different independent aspects which are then combined in the woven program. Weaving is based on specifications, how the different aspects fit together, so-called join-points.

Templates + Filtering

Motivation Consider the creation of simple class skeletons from UML models. UML models are typically stored using the XMI standard [OMG specs], a persistent representation of UML based on the MOF, the Meta Object Facility (see [OMG specs]). XMI files are typically large and complex and contain a lot of information you don't want to when generating your target code. You need to „filter out" the relevant subset of the information in the model, typically the information that can be represented with the help of your target programming language. A typical, simple example would be to generate JavaBeans from an analysis classes that simply have public attributes.

Problem You want to create simple code fragments from higher level specifications. Based on a potentially large and complex description exported from a modelling tool, you have to filter out relevant parts or aspects and create code from them.

Solution Provide a filtering syntax based on pattern matching or similar techniques to „select" specific parts of the model in a declarative way. Then use templates to describe the target code artefacts. The template language must be able to access the selected part of the model to be able to include information from it in the target code.
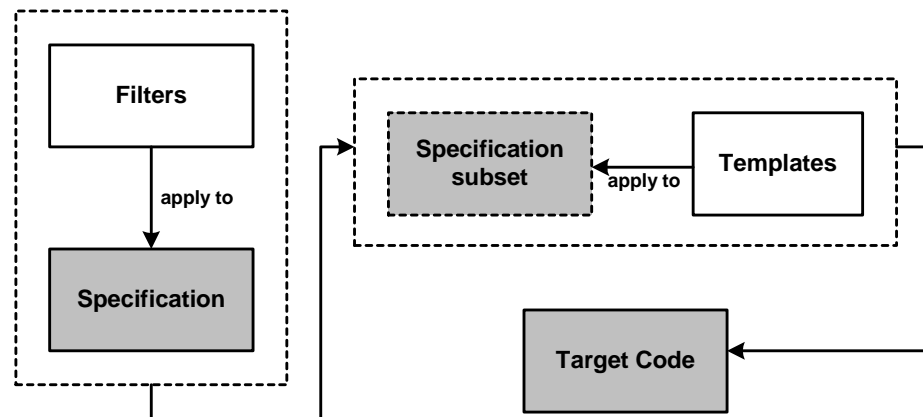
Consequences, Benefits and Liabilities TEMPLATES + FILTERING is an efficient way to perform code generation or model transformation if the filtering mechanism is powerful and the specification well defined. Writing generators in such a scenario is straight forward. It can, however, quickly become too complicated if more than trivial „queries" have to be performed, TEMPLATES + METAMODEL can be a good alternative because it provides a higher-level, specification-syntax independent way to work on the model. A disadvantage of TEMPLATES + FILTERING is that the generation infrastructure is tightly bound to the specification syntax. In contrast to TEMPLATE + METAMODEL there is no indirection level in between. A big advantage of TEMPLATES + FILTERING is that it can be based on industry standard tools such as XML and XSLT.

Motivation revisited Using XSLT as an example, you can use <*xsl:template match...*> clauses to scan the XMI document for relevant information, such as classes and their public attributes. The code to be generated is specified inside the *template* tag. Using <*xsl:value-of ...*> you can access information from the model for inclusion in the generated code. For example, you can access the names of the attributes and generate the respectively named getters and setters in the JavaBean.

Applicability This technique is useful whenever the source specification is highly structured and uses a systematic, well-defined meta syntax. This is especially applicable for model-to-model or model-to-source code generation. XML based systems are often used today for model specification, XMI in particular. To filter, XSLT or XQuery can be used. Source-to-source transformations are typically handled with API-BASED GENERATORS because it is hard to specify rules and filters for source code.

XML + XSLT, Fuut-je [Bronstee, web]

Templates + Metamodel

Motivation Consider a situation where you need to generate code from models in the context of a system family. The family architecture defines a limited set of well-defined architectural building blocks that have a clearly defined meaning in the application domain, and a well-defined mapping onto the implementation platform. Examples could be the three different bean types in EJB, activities, transitions and processes in a workflow system, or the notion of a component in a proprietary component infrastructure, where each component defines (among other things) resources it requires and services it provides. Once you have specified that a model element represents such an architectural building block, the mapping to the implementation is clearly defined and can be automated. The code generation process should be controlled in terms of the domain concepts represented by these building blocks and not in terms of low-level details of the model representation or the to-be-generated code. Also, you want to be able to efficiently encode and enforce domain rules in your generator (for example, service operations that can be invoked asynchronously must feature a *void* return type).

Problem **You want to generate code for products of a software family where you have well-defined architectural building blocks and those have a clearly defined mapping onto an implementation platform. You want to make sure your code generation templates can be specified in terms of domain concepts and do not depend on the low-level details of the modelling data (such as XMI).**
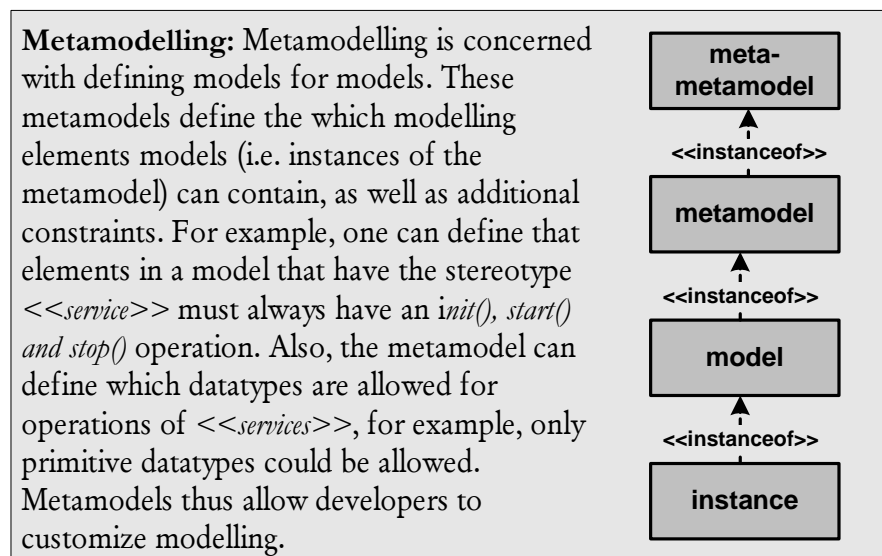
Solution **Provide an explicit, customizable metamodel for your code generator. Implement a two step code-generation process that first instantiates the metamodel based on the model data. Then write the templates in terms of this metamodel. Make sure the metamodel can be adapted to include domain-specific concepts, and that these can be mapped to model elements (such as stereotyped classes)..**
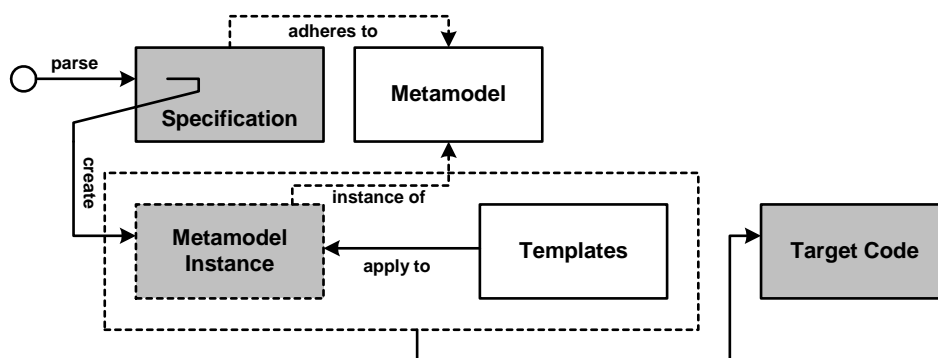
Consequences, Benefits and Liabilities This approach has the big advantage that the parsing of the model and the instantiation of the metamodel is completely separated from the templates

that control code generation. It is therefore easier, e.g. to change the model format without needing to adapt the templates. It is also possible to write the templates in terms of the (possibly adapted, domain-specific) metamodel and not in terms of low-level details of the model representation. This meta-model is adapted to include terms of the problem domain. As a consequence, the templates become more readable and easier to maintain. Also, it is possible to integrate domain specific attributes and behaviour into the metamodel. The metamodel is also an ideal place to include modeling constraints. These will be evaluated by the generator when the metamodel is instantiated during code generation. As a consequence, the template language can be simple and easy to understand, moving complex behaviour into the metamodel (which is implemented with a real programming language).

Metamodelling: Metamodelling is concerned with defining models for models. These metamodels define the which modelling elements models (i.e. instances of the metamodel) can contain, as well as additional constraints. For example, one can define that elements in a model that have the stereotype *<<service>>* must always have an *init(), start() and stop()* operation. Also, the metamodel can define which datatypes are allowed for operations of *<<services>>*, for example, only primitive datatypes could be allowed. Metamodels thus allow developers to customize modelling.

Motivation revisited : Looking again at the custom component infrastructure, the metamodel can provide metaclasses such as *Component* (specializing Class) or *Service* (specializing *Operation*). The *Service* metaclass will verify that each service operation that is tagged as capable of asynchronous execution has a *void* return type, throwing an exception if this rule is violated. Code generation templates can then be formulated with the help of the metaclasses, writing things such as „foreach service in component do < to-be-generated code goes here > ".

Applicability : This approach is typically used during the generation of code from (UML) models where profiles (stereotypes, tagged values, OCL constraints) provide a way to make models mode domain specific.

Known Uses : b+m Generator Framework [b+m, web]], Kenndy Carter iUML [Kennedy Carter, web]

Frame Processing

Imagine building a software system family of stacks. These stacks have the common property that they behave as stacks typically do: you can push elements on top of it, and pop them from there again. However, there are many more or less subtle differences between different stack incarnations: the datatype that should be stacked, optional checking for a maximum size, on-the-fly calculation of the current stack size, thread-safety etc. When creating one of these stacks in the context of your family, you want to be able to assemble a concrete stack (with a specific configuration regarding the above options) from prebuild components based on some kind of abstract specification (e.g. *STACK[type=int, calculateSize=true, threadSafe=false]*. Note, however, that these „components" are not building blocks in the sense of objects. Some features have aspect-like semantics (such as thread safety) whereas others are a kind of parametrization (the element type) and again others require additional, more or less orthogonal functionality (on-the-fly calculation of the current size).

Problem **You want to generate products from a software system family. Each product possesses some valid combination of features defined by the family. These features might be cleanly modularized or they might be crosscutting aspects. You want to implement both of these kinds of features as separate, orthogonal artifacts in order to combine them easily to generate products of a family.**

Solution **Use parametrizable templates called frames. A frame can be seen as a typed function that generates code when evaluated. To compose software systems from a collection of frames, frames have „slots" that can be parametrized with one or more other frame instances, as well as code snippets (such as type names, integrals, etc.). Code generation is controlled by a top-level frame (called the specification frame) - it instantiates, parametrizes and composes instances of the other frames. Finally, it evaluates the composed frame hierarchy in order to generate the code**
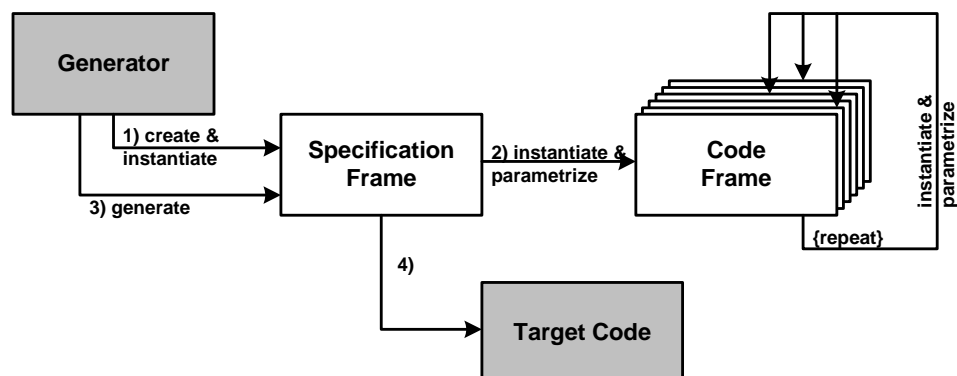
Frame processing comes in two flavours: script-based frame processors work with (typically procedural) scripts that instantiate and parametrize frames (filling in their slot values), building a tree of parametrized frame in-

stances. In contrast, frame processors that are based on the adaptation approach „inject" code into specific locations (i.e. the slots) in other frame instances. The question whether to insert code to slots is controlled by the specification. Because the adaptation-based processors can modify several slot values in different frame instances from within one controlling frame, this approach is well suited for aspect-like composition. Combinations of both approaches are of course also possible.

Note that frame processing, as opposed to TEMPLATES + FILTERING and TEMPLATES + METAMODEL, resembles the imperative style of programming because frames are instantiated and parametrized manually, just as you would instantiate and parametrized a class in an OO language.

To be able to use the „late-code-injection" feature efficiently, an instantiated frame must not be output (i.e. code-generated) directly after instantiation. The instantiated and parametrized frame needs to go into a repository, where it can be reparametrized and modified later, by other frame instances. Only then can aspectual features be realized. In a last step, instantiated frames are code-generated.

Motivation revisited For generating the stacks, you could define a principal frame that generates the *push* and *pop* operations, as well as the stack class itself. This frame has a slot to accept another frame that generates the element data type-specific array declaration to hold the raw data. Another frame is used to generate locking code (for thread-safety). The principal frame has a slot for holding such a thread-safety frame - if one is set, it is used to generate locking code. If none is set, no locking code is generated. This allows us to configure the to-be-generated code by composing frames.

Applicability Frame processing is best used for problems where a small family of closely related members (such as the stacks mentioned above) needs to be generated. Because of the imperative way of programming, generating code from (UML-) models is better done with TEMPLATE + METAMODEL based approaches.

Known Uses Netron Fusion [Netron, web], Delta Software Technology's ANGIE [d-s-t-g, eb], National University of Singapore's XVCL [XVCL, web] (the XML Variant Configuration Language). The initial concept of frames was invented by [Basset, 1996]
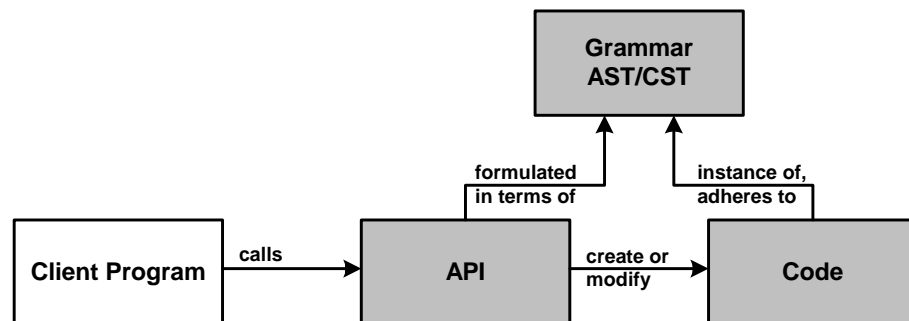
API-based generators

Motivation Object-oriented database systems usually try to provide orthogonal persistence, which means that programming language objects are automatically made persistent (once tagged to be a persistent instance). As a consequence this means that when navigating objects through links (instances of associations) the link target might not yet be loaded into memory, it might be an object stored in the persistent store. It needs to be loaded on-demand when

it is accessed. As we don't want to bother the programmer with manually inserting code to lazy-load the instance, one option is to preprocess the source code (or the bytecode, if available) and insert code at the respective locations that „loads the targetted object if it's not yet in the cache".

Problem **You want to provide a way to generate a small amount of code that needs to handle a well-defined task. You don't have a specification for the generated code, it is always the same, perhaps with some small modifications because it is typically generated in the context of a specialized, problem-specific tool.**

Solution **Provide a code-generation API that is defined in terms of the abstractions of the code to be generated (such as the abstract or concrete syntax tree or the byte code grammar). There are no templates and models, instead a manually written program utilizes the API to create or modify the code in question.**

Consequences,
Benefits and
Liabilities
API based systems are easy to use to generate small amounts of code. Larger model-based generations quickly become overly complex and cumbersome to write, TEMPLATE + METAMODEL is much better suited there. Note also that API-BASED GENERATORS are naturally specific to the programming language they generate, although this fact sometimes get blurred: e.g. generators that generate code for .NET compatible languages can be built in terms of the MSIL while still outputting source code in the various .NET languages. This is because all these languages have the same semantics just using different textual representations (concrete syntax).

Note that API-BASED GENERATORS often serve as a basis for some of the other generator types, such as CODE WEAVING.

Motivation
revisited
Using an API-BASED GENERATOR to add OO-database code, the developer would first parse all persistent classes and those that use instances, creating an AST representation of the code. The developer can then iterate over all expressions that follow links to persistent objects, inserting code (again, using the AST) that ensures the target object of the link is loaded from the database and instantiated in memory.

Applicability
API-BASED GENERATORS are well suited to problems that involve the creation of small amounts of code, typically during postprocessing source code. While there are APIs to access models (such as XMI), most API-based tools modify source code or, even more typically, byte code for platforms

such as Java or .NET. Typical applications can be found in e.g. IDL compilers, AOP tools, OO-databases, etc.

Known Uses   Jenerator (a code generator for Java) [Voelter, web], Compost/Recorder [Uni Karlsruhe, web], BCEL, the Byte Code Engineering Library [Apache, web], .NET's Reflection.Emit [Microsoft, web]

Inline Code Generation

Motivation   Consider developing a threading library that should run on platforms as diverse as Win32, different Unix variants and several real-time operating systems (ACE is such a library that provide threading support, amongst other things). The source code for the library will not run unchanged on all platform. However, you still want to have a single, coherent code-base. As a consequence you might have to make some parts of the code depend on the operating system for which you compile, or some types might need to be adapted to the OS. Also, you might want to be able to optimize some aspects of your source code by e.g. unrolling loops or by using the smallest integer type possible (based on a known, but varying range of values). Again, you want to keep things in one code base, but optimize for specific circumstances.

Problem   **You need to develop source code that is flexible with regards to some aspects, e.g. types used, operating system calls or simply the amount of functionality it provides. All these options should be contained in one code base, selecting the specific options to use at compile time when decisions regarding the options have been made.**

Solution   **Provide a preprocessing facility for your code. This preprocessor can provide include facilities, conditionals, variables, type expressions, etc. that are all evaluated at compile time. The basis for these evaluations can be explicit specifications or the code itself. The generated code is customized and optimized for the specific context resembled by the decisions made during compile time.**
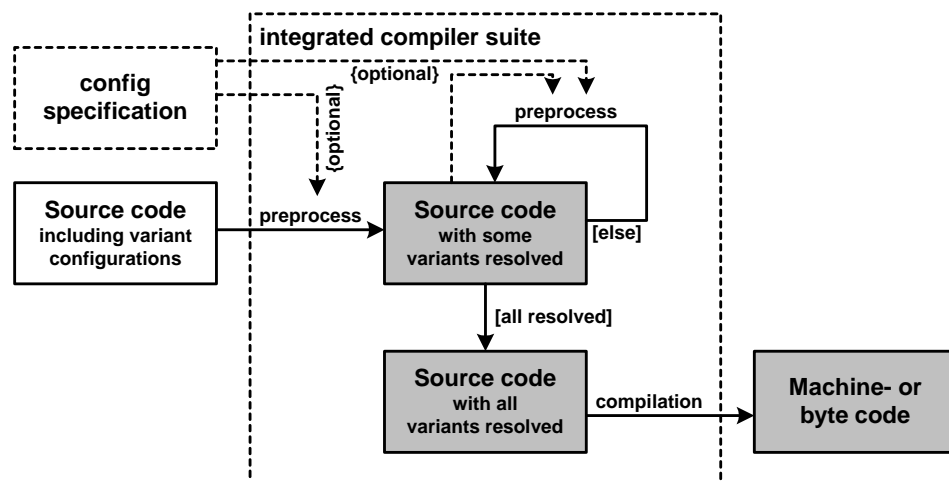
Preprocessing is a powerful concept that is typically used - as in the motivating example - to embed several versions of a source code in on artifact (file). Before actual compilation, these variations are resolved to a specific one. While there is no technical need that this happens in one tool, typically the different preprocessing steps are integrated in one compiler (suite).

There are several flavours of preprocessing. On the one hand there is pure textual replacement based on some kind of specification (such as the C++ preprocessor, being controlled through *#DEFINEs* at compile time) or there is a more integrated approach that takes into account the semantics and type rules of the host language. C++ templates are an example for this approach, the „variant decisions" are based on information in the rest of the source code (maybe parts of the code that are there solely for this purpose).

Note that while there is no limit to the expressive power of such preprocessed languages (the C++ template mechanism is a complete functional programming language), the complexity of the compiler, as well as compile times and resource consumption increases with the power of the preprocessor. This can also be seen as a negative example of the accidental complexity that can arise if several languages are embedded in each other.

ACE uses quite a lot of *#DEFINE* and *#IFDEF* statements to make parts of its C++ source code specific to certain operating systems/compilers/machines. C++ template metaprogramming can be used to define the type of a variable dependent on its maximum value. Static implementations of the Strategy pattern are also very useful to have compiler-checked, but still static variations of functionality. Note that C++ templates are implemented internally simply as (machine) code generation during compilation of the source code whereas preprocessor statements actually modify the source code before it is compiled.

As the practice shows, this method should only be used to configure variants into a code artifact that change only slightly, otherwise the code will become very hard to understand and compile-time increase significantly. For more complex variation problems use CODE WEAVING as it provides „code mangling" based on well-defined semantics.

Include processors for various languages (Pascal, C, C++), the C++ preprocessor, C++ templates [Andrescu, 2001 and Vandervoorde & Josuttis, 2002], Lisp's *'quote* feature [Koschmann, 1990]

Code Attributes

In the context of Enterprise JavaBeans development, a bean consists of several artifacts: the bean implementation class, the bean's various interfaces and an XML configuration file called the deployment descriptor [Voelter et. al. 2002]. These artifacts are related to each other, for example all operations defined in the bean's remote interface have to have a corresponding

method in the bean's implementation class, and all operations in the interface need to have a couple of security and transaction properties defined in the deployment descriptor. To be able to deploy the bean into an application server, all these different artifacts need to be consistent. Keeping these things consistent manually is rather hard to do and the Java compiler cannot help you either. Ideally, you only want to implement one artifact - the implementation class - manually and the other ones should be generated automatically, as far as possible.

Problem **You have a piece of source code and you need to generate additional artifacts based on the code and additional information. While most of the information you need for the generation is available in the source code, you need to specify additional information that cannot be directly expressed with the means of the language, but is closely related to the information in the code.**

Solution **Annotate the source code with additional attributes. In most languages these attributes will be special comments. The code generator parses the code and the comments and creates the additional artifacts based on this information.**
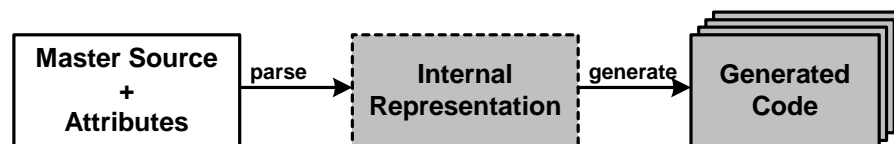
Consequences, Benefits and Liabilities This pattern typically uses API-BASED GENERATORS for its own implementation. The big benefit of this pattern is that you can annotate code with whatever additional information you need as long as the subsequent code generator „understands" the annotations. Because the generated code is „deduced" from the source code and its attributes, the different artifacts cannot easily get out of sync.

Implementation of the attributes varies based on the language/system in use. In Java, these attributes are typically specified as special comments beginning with a @ (the same way as JavaDoc does it). In .NET, it is possible to define custom attributes (which are classes themselves) that are compiled and included in assemblies (.NET's packaging and distribution artifacts). The attributes can then be accessed using reflection, and code can be generated based on this information (for example using CodeDOM or Reflection.Emit, see [Microsoft, web]).

Motivation revisited Using XDoclet, you can provide special JavaDoc comments in your bean implementation class that are subsequently used by the XDoclet code generator to define the bean's remote/local interface, it's (local) home interface, as well as deployment descriptors for different application servers. By including the XDoclet execution in the build process, you can easily make sure that the different EJB artifacts are always in sync.

Applicability This pattern should be used whenever there is one „master" artifact (source code) and several other artifacts have to be generated, depending on the master. Attributes are then necessary to support additional information not expressible with code only.

Known Uses JavaDoc is one of the most well-known tools that use this approach, although they don't generate Java source, but documentation HTML. XDoclet provides a framework where you can define your own „special comments", tags, and generate artifacts based on these. Among other things, it provides tags to solve the EJB problem mentioned in the motivation section. Microsoft's .NET [Microsoft, web] has attributes as first-class citizens. You can define your own attributes and attach them to methods, classes, etc. They can be accessed using reflection.

Code Weaving

Motivation Consider the construction of an application that displays the state of a set of objects graphically. To separate the data model (the „business objects") from the display, the well-known model-view-controller pattern can be used. However, while it does help to separate rendering code from the data model, introducing the rendering layer still needs an invasive change into the model classes; the *updateView()* operation needs to be called at appropriate times in the model code. And the model needs to have a reference to its view (or some kind of mediator) to be able to notify the renderer of changes in its state. Changing the policy that determines when and how often the view is updated might required additional adaptations to the source code. This is especially problematic since, while these changes are highly mechanic, they are distributed all over the source code; they cannot easily be localized to one specific location in the code.

Problem **You want to provide a way to join different code artifacts in a well-defined manner. The artifacts to join may just implement different optional application features that need to be plugged into an application in a controlled way, or they might address different, often orthogonal concerns, or aspects of a system that cross-cut the primary decomposition hierarchy.**

Solution **Define different meta-artifacts from which programs can be composed. Clearly define the semantics and responsibilities of each meta-artifact. Key to composing artifacts in applications is then a precise definition of how the different (meta-)artifacts go together and how they influence each other. A code-weaver will then join the different artifacts according to these definitions.**
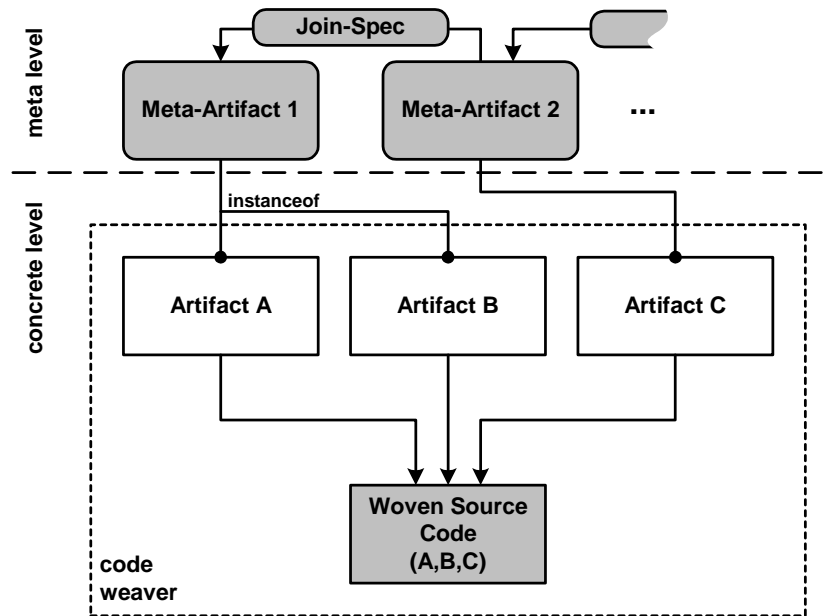
Consequences, Benefits and Liabilities This pattern describes a general way of weaving pieces of code together. Such pieces of code can be localized or cross-cutting. The important thing is that the relationship of the different code artifacts must be clearly defined.

The most prominent member of this kind of tools is AspectJ, a code-weaving based tool for aspect-oriented programming in Java. Note, however, that AspectJ can also be used as a straight-forward code-generator, not using many of the AOP-concepts. Another tool is IBM's Hyper/J which allows composition and extraction of different code artifacts on Java byte-code level.

This pattern might sound like „The Great AOP Pattern". However, aspect-oriented programming is not the same: semantically, AOP [AOSD, web] is a part of this pattern, other ways to weave code can be used. And technically AOP need not necessarily be realized with code generation, there are several completely dynamic AOP frameworks (such as Aspect/S Squeak/Smalltalk).

Motivation revisited The model-view-controller problem introduced in the motivation section is a typical example for a cross-cutting concern, addressed by AOP[1]. Using a aspect weaver, you can define an aspect that introduces an association to a view object into the model classes. You can also advice all the locations in the code[2] that should trigger the a view update. All these specifications are done in one separate   artifact (the aspect) and are woven into the normal program code by the aspect weaver.

---

1. The example is actually adopted from the AspectJ tutorial.
2. Actually, you advice points in the execution of the program. But for the sake of simplicity, assume that these correspond to specific locations in the code (which they actually do in the vast majority of the cases).

The code weaver will need some kind of metamodel to work correctly, typically this metamodel consists of the syntax/semantics of the languages that are woven. Developing this metamodel and the code-weaver is a lot of work and is certainly not done in an ad-hoc fashion. You should only invest in such an approach if you can frame your code generation problem very well and if you can specify a well-defined semantic relationship among the different artifacts. Then, however, using this approach is very powerful because the weaver can give you very expressive error messages even at compile (or weave-) time. Once the weaver completes without errors, the generated, woven code is very likely to be correct in the sense that it does what the metamodel specifies.

Known Uses AspectJ [AspectJ, web], Hyper/J [Hyper/J, web]. Some metamodelling tools such as GME provide a framework for such an approach [ISIS, web]. Note that compile-time MOPs such as Kava [Kava, web], OpenC++ [OpenC++, web] can also be seen as an instance of this pattern.

Integrating generated code with non-generated code

After having looked at the different patterns for code generation we now look at how generated and non-generated code can be integrated. This is important, since often, one can hear statements such as „I don't like code generation because the generated code is hard to read and modify!". Typically, this is true. The problem is not so much with layout, formatting and style - modern generators can create nice looking code. However, it is not easy to actually *understand* the generated code, because you don't want to deal with the low-level details of the generated code.

Two kinds of generators In most cases, it should never be necessary to read, let alone modify generated code. There are other ways of successfully integrating generated and non-generated code. To get started, let's first classify the patterns into two groups with regards to whether the generated code is a separate artifact or whether it is seamlessly integrated. The following patterns fall into first category:

- TEMPLATES + FILTERING
- TEMPLATES + METAMODEL
- FRAME PROCESSING
- API-BASED GENERATOR
- CODE ATTRIBUTES

The second group of patterns are those where the pattern itself mandates a specific way of integration of generated and non-generated code:

- INLINE CODE GENERATION: here the generated code becomes a part of the non-generated code during compilation.
- CODE WEAVING produces a „merger" of the different source artifacts as the result of the weaving process.

This distinction is necessary because following discussion is only applicable to code generation techniques where generated and non-generated code are two different artifacts, the first category. The following illustration provides an overview of how to integrate generated and non-generated code in an object-oriented language and is explained below.
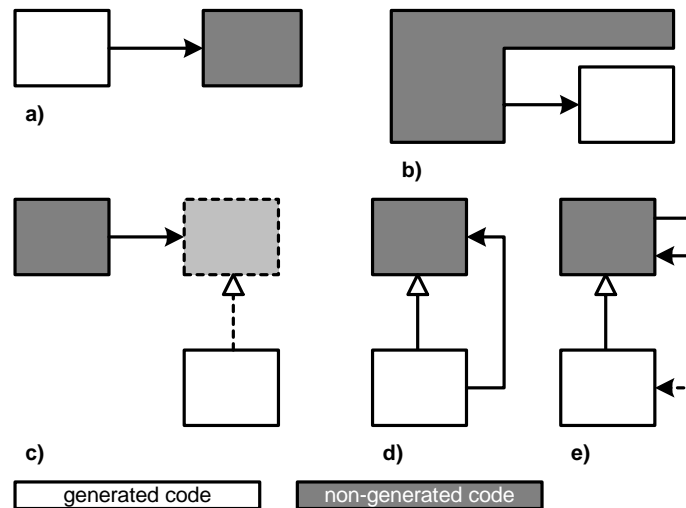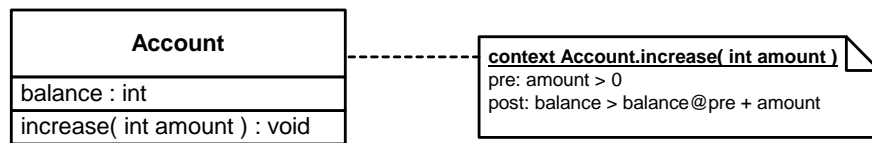

First of all, generated code can call non-generated code contained in libraries (case (a)). This is an important use, as it basically tells you to generate as few code as possible and rely on pre-implemented components that are used by the generated code. As shown in (b), the opposite is of course also possible. A non-generated framework can call generated parts. To make this more practicable, non-generated source can be programmed against abstract classes or interfaces which the generated code implements. Factories [Gamma, et. al. 1994] can be used to „plug-in" the generated building blocks, (c) illustrates this.

Generated classes can also subclass non generated classes. These non-generated base classes can contain useful generic methods that can be called from within the generated subclasses (shown in (d)). The base class can also contain abstract methods which it calls, they are implemented by the generated subclasses (template method pattern, shown in (e)). Again, factories are useful to plug-in instances.

Of course it is possible to combine several of the integration techniques explained here. Generated code can use a non-generated abstract interface that is implemented by another piece of generated code and accessed through a non-generated factory. And in the context of a project, different parts can be implemented using different means.

There is one special case I want to look into, that is typically considered a reason to modify generated code. Imagine the following UML model:

```
+--------------------------------+          +-----------------------------------------+
|           Account              |          | context Account.increase( int amount )  |
+--------------------------------+ -------- | pre: amount > 0                         |
| balance : int                  |          | post: balance > balance@pre + amount    |
+--------------------------------+          +-----------------------------------------+
| increase( int amount ) : void  |
+--------------------------------+
```

You might want to use a generator that generates class skeletons, including the pre- and postconditions. You want to ensure that the pre- and postconditions are always checked, no matter what the developer enters in their implementation code. A simplistic solution could be the following, using Java:

```
// generated
class Account {
  int balance;
  public void increase( int amount ) {
    assert( amount > 0 ); // precondition
    int balance_atPre = balance; // postcondition
    // --- protected area begin ---
    // --- protected area end ---
    assert( balance = balance_atPre + bamount ); // postcond.
  }
}
```

The problem here is that using simple inheritance does not work. You could check preconditions by calling *super.increase(amount)* at the beginning of the implementation subclass's increase method. But developers could forget to do so. And checking the postcondition that way is not possible. You could also genereate operations such as *increase_pre(...)* and *increase_post(...)* requiring developers to call these at the beginning of the method in their implementation subclass. But again, they could forget to do so. The easiest solution is to use a slight modification ofh the template method example above, using method renaming:

```
// generated
public abstract class Account {
  int balance;
  public final void increase( int amount ) {
    assert( amount > 0 ); // precondition
    int balance_atPre = balance; // postcondition
    increase_internal(amount)
    assert( balance = balance_atPre + bamount ); // postcond.
  }
  protected abstract void increase_internal(int amount);
}
```

Developers would the overwrite the internal method in their subclass and use a factory to make sure that *Account* instances are actually instances of the implementation subclass, for example *AccountImpl*.

```
// generated
class AccountImpl extends Account {
  protected void increase_internal(int amount) {
    balance += amount;
  }
}
```

Because the external interface used by clients is still the *increase* operation, pre- and postconditions are always checked. There is no way for clients or implementation developers around that.

As a general guideline, one can say that integration of generated and non-generated code can be simplified by defining a clear architecture with well-defined responsibilities and interfaces, which also defines which parts are generated, and which are not. The Small Components Project [SmallComponents, web] is an example of such an approach.

How these patterns fit together

This section shows the relationships among the patterns explained above. Let us first look at an overview:
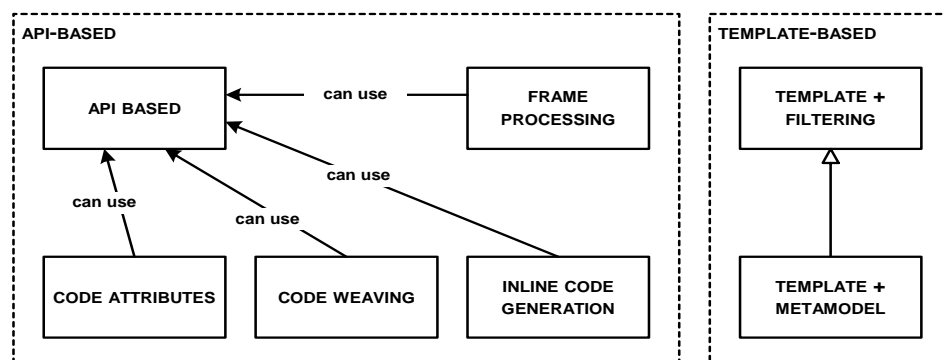
Template based approaches

There are two fundamentally different basic approached: template-based generation and API-BASED GENERATORS. TEMPLATES + FILTERING generates code directly from the model specification and templates, combined by filters and rules. TEMPLATES + METAMODEL can be seen as an extension of TEMPLATES + FILTERING since the metamodel and its instances are „put in the middle", to decouple the templates from the model syntax.

API-based techniques

In the realm of API-based approaches, the direct API-BASED GENERATORS is obviously the most fundamental pattern. CODE ATTRIBUTES is mainly concerned with how generation information is specified, namely in terms of code attributes. However, it does not say anything about how the target code is generated. In most tools (.NET, XDoclet) an API-based approach is used. CODE WEAVING is mainly concerned with how the different aspects are specified and how they can be joined rather than saying how the weaving should take place. Typically, the different code artifacts are parsed, woven on AST level using an API-based approach and then unparsed. An alternative approach is of course based on pure text processing using regular expressions and the like. INLINE CODE GENERATION can also use API-based approaches, although typically, simple text processing and „printf-generation" is applied. Last but not least, FRAME PROCESSING can also use API-based approaches; internally, frame instances are based on the syntax/metamodel of the target language.

The following table presents some categorization/overview over the patterns in this language.

	generated/ ungenerated code	template / API	initial learning complexity	suitability for complex uses	flexibility
Template + Filtering	separate	template	simple	not very good	not very good
Template + Metamodel	separate	template (plus m-model API)	high	very good	very good
Frame Processors	separate	template	high	very good	very good
API-based	separate/ integrated	API	simple/high	depends on abstraction level of API	depends on abstraction level of API
Inline Code Generation	integrated	template/API	simple	not very good	good
Code Attributes	separate	template/API	simple	good	not very good
Code Weaving	integrated	API	high	good	good

Example Technologies

After discussing the different patterns in an abstract fashion above, let's now look at some concrete examples. Note that because of space limitations, we cannot go into too much detail here. For more detailed examples see the slides on code generation at my web page ([Voelter, slides]).

Based on the overview of the code generation applicabilities in the application development lifecycle from the introduction, we now show where and how these technologies are used today. The following table presents an overview.

	Model Transformation	(Source) Code Generation	Code Transformation	Byte Code Creation/Modif.
Template + Filtering	XML+XSLT	XML+XSLT Fuut-je		
Template + Metamodel		b+m Generator Framework	OpenC++ OpenJava	
Frame Processors		ANGIE XVCL		
API-based	XMI APIs	Jenerator	Compost	BCEL .NET CodeDOM, Reflection.Emit
Inline Code Generation			Preprocessors C++ Templates	
Code Attributes		XDoclet		.NET
Code Weaving			AspectJ AspectC++	HyperJ

We will now provide a mapping from the patterns to concrete technologies and also show some very brief examples of their use.

Examples for Templates + Filtering

The simplest example for the pattern is to use XML-based specifications and XSLT-based filtering. In this case, the XSLT contains source code as the text output. The usual XSLT tags specify the filers, or application rules. The following piece of source code specifies, in XML, a *Person* class with two attributes *name* and *age*:

Specification
```
<class name="Person" package="de.voelter.test">
   <attribute name="name" type="String"/>
   <attribute name="age" type="int"/>
</class>
```

The product of the code generation should be a Java class that looks like the following:

Resulting Code
```
package de.voelter.test;

public class Person {
   private String name;
   private int age;
   public String get_name() {return name;}
   public void set_name( String name) {this.name = name;}
   public int get_age() {return age;}
   public void set_age( int age ) {this.age = age;}
}
```

To generate code from this specification, one can create an XSLT document that selects each *class* tag from the XML document and creates a Java class body, then going on to select each *attribute* and generating a Java mem-

ber variable as well as getter and setter operations. Java source code snippets are printed bold.

XSLT Generator Template

```
<xsl:template match="/class">
  package <xsl:value-of select="@package"/>;
  public class <xsl:value-of select="@name"/> {
    <xsl:apply-templates select="attribute"/>
  }
</xsl:template>

<xsl:template match="attribute">

  private <xsl:value-of select="@type"/>
    <xsl:value-of select="@name"/>;

  public <xsl:value-of select="@type"/>
    get_<xsl:value-of select="@name"/>() {
       return <xsl:value-of select="@name"/>;}

  public void set_<xsl:value-of select="@name"/> (
    <xsl:value-of select="@type"/>
     <xsl:value-of select="@name"/>) {
        this.<xsl:value-of select="@name"/> =
          <xsl:value-of select="@name"/>;  }

</xsl:template>
```

As can be seen even from this simple example, the XSLT syntax is not very readable, even for simple problems this approach becomes complex quickly. And usually, you'll need a pretty printer to clean up and correctly indent the generated code. Also, you have to deal with some XML processing issues, for example, you cannot simply use a *space* because that is ignored by XML/XSLT, you might need to use XML entities. The biggest advantage of this approach is that you don't need special tools. For more details, see Craig Cleaveland's book [Cleaveland, 2001].

## Examples for Template + Metamodel

The Model   Let's look at the following example from a system that generates component infrastructures for embedded systems. The following is an example model a system consisting of a couple of components and uses the b+m Generator Framework [b+m, web[.
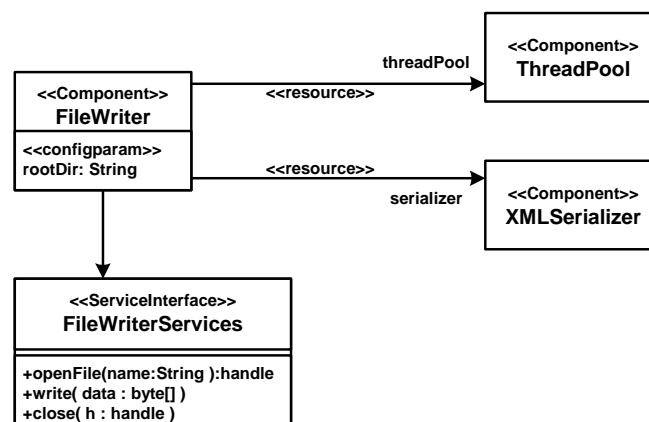
Note the custom-stereotypes applied to the classes. These stereotypes have a well-defined semantics in the respective domain:

- A component is a special kind of class that is instantiated an controlled by a container.

- A component has config parameters that need to be supplied by a config file when the container starts up.

- A component provides services in the form of a set of operations.

- And a component requires resources in order to run properly. The container needs to provide them when it starts up.

Metamodel   The next illustration shows the custom-made metamodel that represents these components and the relationships. The metaclasses are subclasses of the standard MOF metaclasses available in the generator tool.
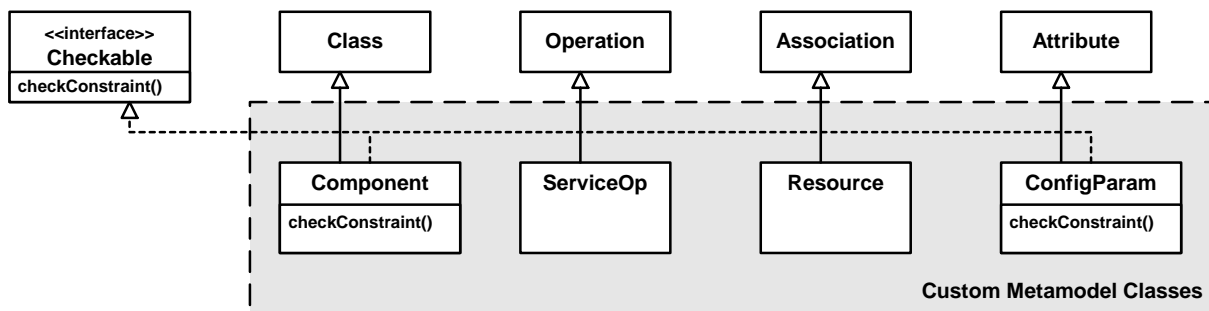
When the parser analyzes the model (it is represented as an XMI file) it instantiates the metamodel classes based on the stereotypes. The metaclasses shown above can contain custom functionality such as constraint checking. We provide the interface *Checkable* in the metamodel. The following shows the code that needs to be implemented to ensure that *ConfigParams* can only be attributes of type *String*.

```
public class ConfigParam
        extends Attribute implements Checkable {
   public void checkConstraint() {
     if ( getType() != Type.STRING ) {
       throw new DesignError(„ConfigParams have to"+
                              „be Strings");
     }
   }
}
```

We can now write templates that generate code by iterating over the metamodel an accessing properties of the metaclasses. The following is a simple template. The template control language code is rendered in bold.

Template   ```
<<FOREACH Component AS c IN Model {>>
   public class <<c.Name>> extends ComponentBase {
     <<FOREACH ConfigParam AS p {>>
       private String <<p.Name>>;
         public void configure_<<p.Name>>(String p) {
         this.<<p.Name>> = p;
       }
     <<}>>
   }
<<}>>
```

It is ineresting to see that, in the template, we access the *Name* property of the *Component* metaclass, as well as the *ConfigParams* property. These need to be defined in the corresponding metaclass to be accessible from within the templates. The following shows an example definition of the *Component* metaclass.

```
public class Component extends Class {
  public String getName() {
    return super.getName();
  }
  public Collection getConfigParams() {
    // get all attributes
    // check which of them are instanceof ConfigParam
    // return these
  }
}
```

Example for Frame Processing

This example uses the ANGIE processor [d-s-t-g, web]. Consider the generation of simple member variables into a system. The declarations should in the end look somewhat like the following:

```
short int aShortNumber;
```

This little piece of code contains already a significant number of variable aspects, such as the name of the variable, the type (short, int, long) depending on the required range of values, an optional initialization parameter, etc. The following piece of code shows a frame that takes care of this:

A Frame
```
.Frame GenNumberElement(Name, MaxValue)

   .Dim vIntQual = (MaxValue > 32767) ? "long" : "short"
  .Dim sNumbersInitVal

  <!vIntQual!> int <!Name!> <? = <!sNumbersInitVal!>?>;

.End Frame
```

The first line declares the frame - this is basically a constructor with two parameters, the name of the number element and the max value. Line two determines based on the max value if we need a *short* or a *long int*. Line three defines a globally accessible slot (whose value is supplied optional by the current frame's environment when it is instantiated) whereas the fourth line is the so-called host-code that contains embedded (target) code and a (optional) slots: The <!...!> syntax accesses the content of the slot inside it whereas the <?...?> generates the code only when the slot inside actu-
Frame ally has a value. The following statement instantiates the frame:
Instantiation
```
.myNumbElm = CreateFrame("GenNumberElement",
            "aShortNumber", 100)
```

Note that this does not yet create any code. It only instantiates the frame and stores it in the internal frame instance repository. For example the value of the *sNumbersInitVal* slot is not yet bound and can still be changed!

When calling

```
.Export(myNumbElm)
```

the instantiated frame is exported. Instead of exporting it (which actually generates the source code), this frame instance can also be used as a value for slots in other frame instances, building a hierarchy of instantiated frames. For example, consider the following frame that generates a Java class, Java code in bold again:

```
.Frame ClassGenerator(fvClassName)
  .Dim fvMembers = CreateCollection()
  public class <!fvClassName!> {
     <!fvMembers!>
  }
.End Frame
```

This frame takes the class name as a parameter and also defines a multi-valued slot inside it (a „member"). An external script, or another frame, can now add values to this collection. For example, the number elements from before.

```
.Static myGeneratedClass As ClassGenerator

.Function Main(className)
  .myGeneratedClass =
     CreateFrame(„ClassGenerator",className)
  .Add(myGeneratedClass.fvMembers,
     CreateFrame(„GenNumberElement","i", 1000))
  .Add(myGeneratedClass.fvMembers,
     CreateFrame(„GenNumberElement","j", 1000000))

.End Function
```

Exporting *myGeneratedClass* will output a Java class with two members, i and j.

Examples for API-based generators

Let's first take a look at the .NET code generation facility, CodeDOM. Consider generating the following C# code:

```
public class Vehicle : object {
}
```

This piece of code is generated by the following piece of (yet another) piece of C# code:

```
CodeNamespace n = ...
CodeTypeDeclaration c = new
   CodeTypeDeclaration („Vehicle");

c.IsClass = true;
c.BaseTypes.Add (typeof (System.Object) );
c.TypeAttributes  = TypeAttributes.Public;
n.Types.Add(c);
```

The generating C# is actually still independent of the language in which the generated code is written as it operates on top of the semantics of the MSIL. Outputting of the code is done with an instance of the *ICodeGenerator* interface which can be implemented for various languages. The generating

program can even determine which language features are supported by the targetted source language:

```
ICodeGenerator cg = // obtain code generator for
                    // intended target language
if (cg.Supports(GeneratorSupport.
    MultipleInterfaceMembers)) {
  ...
}
if (cg.Supports(GeneratorSupport.DeclareValueTypes)) {
  ...
}
```

Outputting the code is also very simple. The following operation is provided by the code generators:

```
void GenerateCodeFromNamespace( CodeNamespace e,
    TextWriter w, CodeGeneratorOptions o );
```

.NET Reflecton.Emit Another way of generating code in .NET is using the Reflection.Emit package. The nice thing about this one is that it does not generate source code, instead it directly creates MSIL code in the form of readily executable assemblies. .NET uses this feature extensively to on-the-fly generate client-side proxies for remote objects. Let's look at the following piece of C# code. First you need to construct the name for the assembly:

```
AssemblyName an = new AssemblyName();
an.Name = „MyOwnAssembly";
```

Then you create the assembly itself, in the current application domain:

```
AssemblyBuilder abuilder =
    Thread.GetDomain().DefineDynamicAssembly(an,
                    AssemblyBuilderAccess.Save);
```

Once this is done, you can create a module in the assembly and create a type, here we create a class called *ExampleClass*.

```
ModuleBuilder module = abuilder.
    DefineDynamicModule(„Example",„Example.DLL" );
TypeBuilder myClass = module.
    DefineType(„ExampleClass", TypeAttributes.Public);
```

You can then subsequently add a method to this class, in this case one that takes two *ints* and returns another one.

```
Type[] params = new Type[2];
params[0] = typeof(int);
params[1] = typeof(int);
Type returnType = typeof(int);
MethodBuilder addMethod = myClass.DefineMethod
    („Add",MethodAttributes.Public | MethodAttributes.
        Virtual,returnType,paramTypes);
```

Last but not least you can also add real implementation code - however, this needs to happen on the IL level, not with source code:

```
ILGenerator ilg = addMethod.GetILGenerator();
ilg.Emit(OpCodes.Ldc_I4, 0);
....
```

Examples for Inline Code Generation

The following examples are probably really well known, they cover the C/C++ preprocessor. The most well-known statement is probably the *#include* statement which textually includes other files:

```
#include „iostream.h"
```

The preprocessor can also be used for conditional evaluation of code, as in the following example: (taken from the ACE libraries)

```
#if defined (ACE_HAS_TLI)
static ssize_t t_snd_n (ACE_HANDLE handle, const void
   *buf, size_t len, int flags, const ACE_Time_Value
   *timeout = 0, size_t *bytes_transferred = 0);
#endif /* ACE_HAS_TLI */
```

The above piece of code is only compiled if the *ACE_HAS_TLI* flag is defined (which can be interpreted as consituting a *true* value whereas undefined can be seen as *false*). This is either done in another part of the code using a

```
#define ACE_HAS_TLI
```

statement, or as a define that is passed to the compiler. However, the C/C++ preprocessor can also be used for defining constants as in

```
#define MAX_ARRAY_SIZE 200
#define AUTHORNAME MarkusVoelter
```

or to define simple macros that are evaluated at compile time. Note that there is no syntax checking available when defining the macro, it is simply textually expanded. The compiler then syntax-checks the resulting code.

```
#define MAX(x,y) (x<y ? y : x)
#define square(x) x*x
```

A much more powerful means for inline code generation, again in C++, is based on template metaprogramming. Here the fact is exploited that the C++ template evaluation mechanism is a full-fledged functional programming language (with an awkward syntax, though). For example, the following templates constitute a compile-time *if*.

```
template<bool condition, class Then, class Else>
struct IF {
  typedef Then RET;
};
//specialization for condition==false
template<class Then, class Else>
struct IF<false, Then, Else> {
  typedef Else RET;
};
```

This can be used, for example, to determine the type of an expression at compile time.

The following is a (admittedly useless) example:

```
const int a = 10000;
const int b = 1200;

void main() {
   IF<(a+b<32000), short, int>::RET i;
   i = a+b;
}
```

This automatically uses the smallest integer type possible to hold the sum of *a* and *b*. Note that changing the values of a and b automatically adjusts the type of i to be large enough to hold the sum. Using this technique, you can build compile-time linked lists, provide compile-time type information, etc. For details see Andrescu's book.

A last example is the calculation of a factorial at compile time.

```
struct Stop
  { enum { RET = 1 };
};


template<int n>
struct Factorial {
  typedef IF<n==0, Stop, Factorial<n-1> >::RET
       PreviousFactorial;

  enum { RET = (n==0) ? PreviousFactorial::RET :
       PreviousFactorial::RET * n };
};
void main() {
  cout << Factorial<3>::RET << endl;
}
```

Examples for Code Attributes

XDoclet   Let's first look at the XDoclet example. The following is a bean implementation class (a stateless session bean):

```
/**
 * @ejb:bean type="Stateless"
 *           name="vvm/VVMQuery"
 *           local-jndi-name="/ejb/vvm/VVMQueryLocal"
 *           jndi-name="/ejb/vvm/VVMQueryRemote"
 *           view-type="both"
 */
public abstract class VVMQueryBean

    /**
     * @ejb:interface-method view-type="both"
     */
    public List getPartsForVehicle( VIN theVehicle ) {
        return super.getPartsForVehicle( theVehicle );
    }
}
```

The *@ejb:bean* tag in the comment area for the class specifies that this class should be a Stateless Session Bean, its name should be *vvm/VVMQuery* and the name of the remote home interface in JNDI should be */ejb/vvm/VVM-QueryRemote*. The *view-type* attribute specifies that the bean should have a local

and a remote interface. In the method's comment, we specify that this method should be available in the interface(s) of the bean. *view-type="both"* specifies that it should be visible in both interfaces.

Running the XDoclet tool agains this class source generates the remote interface, the local interface, the home interface and the XML deployment descriptor. Running the tool can be integrated into Java's *ant* build tool, an automated generation of the code as part of the build process is possible with this approach.

Writing custom tags (or attributes) is rather simple. The nice thing about XDoclet is that is implemented on top of JavaDoc. JavaDoc handles all the parsing of the code and the attribute extraction.

.NET In the .NET world, it is possible to attach attributes to interfaces, classes, members, and methods. .NET uses this facility for several „internal" uses, for example, classes can be marked to be *Serializable*, or a method in a class can be marked to be accessible as part of a webservice (*Webmethod*).

Associating an attribute with a code entity you just specify the attribute in the code. The following example declares the class *SomeClass* to be serializable.

```
[Serializable]
public class SomeClass {
  // ...
}
```

Another example is the webmethod-declaration:

```
public class MyWebservice : System.Web.Services.WebService {
  public void init() {
    // ...
  }
  [WebMethod]
  public int add( int a, int b ) {
    return a+b;
  }
}
```

It is easily possible to define your own attributes. You have to inherit from the *System.Attribute* class and specify the *AttributeUsage* attribute for the class:

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple=false)]
public class Transaction : System.Attribute {
  private string txType;
  public Transaction( string txtp ) {
    txType = txtp;
  }
}
```

You can now use this attribute on all methods (*AttributeTargets.Method*), such as here:

```
public class MyComponent : ComponentBase {
  [Transaction(„required")]
  public void someMethod( ... ) {
    // ...
```

```
      }
   }
```

At runtime, e.g. a container that runs the above component, can use .NET reflection to find out about the value of the Transaction's *txType* attribute. The container can then use .NET's Reflection.Emit facility to code-generate a proxy than handles transactions.

Examples for Code Weaving

AspectJ   Let's look at the simplest of all AspectJ examples, namely the introduction of logging output into normal Java code. The following is the aspect that defines this an aspect that prints out log messages whenever an operation on

```
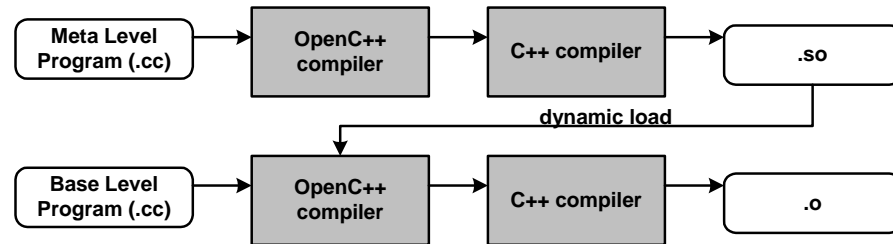aspect Logger {
   public void log( String className, String methodName ) {
    System.out.println( className+"."+methodName );
   }
   pointcut accountCall():  call(* Account.*(*));
   before() calling: accountCall() {
     log( thisClass.getName(), thisMethod.getName() );
   }
}
```

This aspect specifies that before the calling of all operations on the *Account* class. After running the aspect weaver with the *Account* (and other) classes, the following code results:

```
public class SomeClass {
   private Account someAccount = ...;
   public someMethod( Account account2, int d ) {
     System.out.println( „SomeClass.someMethod" );
     someAccount.add( d );
     System.out.println( „SomeClass.someMethod" );
     account2.subtract( d );
   }
   public void anotherMethod() {
     System.out.println( „SomeClass.anotherMethod" );
    int bal = someAccount.getBalance();
   }
}
```

OpenC++   Another example of code weaving is OpenC++ [OpenC++, web]. This is an example of a compile-time meta object protocol (MOP). It is important to emphasize „compile-time", because typically, MOP's are implemented dynamically as a means for reflection [Kiczales et. al., 1991] and do not use any code generation at all. In principle, the OpenC++ compiler works the following way: first you write a meta-level program, which specifies how to translate or analyze a C++ program. It is written in C++. Then the meta-level program is compiled by the OpenC++ compiler and linked to the compiler itself. The resulting compiler translates or analyzes a source program as the meta-level program specifies. The meta-level program is written conforming to the OpenC++ MOP which exposes the in-

ternal structure of the compiler with object-oriented abstraction. The following illustration shows this process.

Take a look at the following OpenC++ (base-level) source code:

```
class Point {
   ...
};
metaclass MyMetaClass Point;
```

The important statement - and the only difference to ordinary C++ - is the one in the last line, where the metaclass *MyMetaClass* is assigned to the class *Point*. To make this program translatable, we have to define the metaclass. This class contains operations that specify how the compiler translates the base-level source. For example, the following operation (which could be a member of *MyMetaClass*) renames all members of its instances (i.e. base-level classes with *MyMetaClass* associated as metaclass) which are called *f* to *g*.

```
void MyClass::TranslateClass(Environment* e) {
  Member m;
  LookupMember("f", m);
  m.SetName(Ptree::Make("g"));
  ChangeMember(m);
}
```

Several other utility functions can be used to alter the instances of the current metaclass, such as *RemoveClass()*, *ChangeName(Ptree* new_name)*, *Append-BaClass(Class* c, int specifier = Public)* or *AppendMember(Member& added_member, int specifier = Public)*.

We can also work on a different level of granularity, e.g. on the level of method bodies. The following operation sets all method bodies to be empty:

```
void MyClass::TranslateMemberFunction(Environment* env,
                                      Member& m) {
   m.SetFunctionBody(Ptree::Make("{}"));
}
```

Usage Examples

This section just briefly wants to introduce some concrete example projects where the respective technologies have been used. I cannot provide examples for all technologies, so if you know of other examples, **please contact me so that I can integrate them here.**

TEMPLATES+ FILTERING I have no concrete project examples for this approach, I know, however, that several people and companies are working with XML+XSLT. Also, the FUUT-je tool [Bronstee, web] has been used in real projects at IBM, as far as I know.

TEMPLATES+ METAMODEL The b+m Generator Framework [b+m, web], as an example of the TEM-PLATES + METAMODEL approach has been used by b+m in several projects in a J2EE context, typically generating EJBs from models. I am using this approach for the automatic generation of component containers [Voelter, et. al., 2002] in a project about component infrastructures for embedded systems [SmallComponents, web]. The approach is also currently evaluated in the context of the FI 2010 initiative [FI2010, web] for the generation of IDL-like interfaces, as well as dummy implementations and CORBA server apps from UML models.

FRAME PROCESSING I have no first hand experience with frame based technologies. As long as nobody told me about projects using it (I know many exist, I just don't know the people personally), please look at the homepages of the respective tools, they have case studies and success stories online: [d-s-t-g, web], [Netron, web], [XVCL, web].

API-BASED GENERATORS Let's pick three examples. The Jenerator tool [Voelter, web] has been used in several project, for example to generate proxy implementations (for security checking, e.g.) based on Java interfaces. The BCEL library is widely used throughout the industry, e.g. it is used in the Xalan XSLT processor, the Kava reflective Java implementation, the Ozone OO database, AspecJ, and more. For imformation look at the [BCEL projects, web] page. The .NET Reflection.Emit package is used for example by the .NET CLR itself to on-the-fly-generate remoting proxies. Last but not least, the RECORDER tool [Uni-Karlsruhe, web] is used in an EAOP tool.

INLINE CODE GENERATION The C/C++ preprocessor is used everywhere, anyway. A particularly extensive example can be found in the ACE libraries [Schmidt, web]. Template metaprogramming can be seen put to great use (not just explained, also used!) in Andrescu's book [Andrescu, 2001], inside the Boost library [BOOST, web] or as part of the Generic Matrix Computation Library [GMCL, web].

CODE ATTRIBUTES Attributes are used everywhere in .NET, for example in the context of remoting, webservices, serialization, etc. Tools such as XDoclet [XDoclet, web] are used to generate EJB artifacts, or mapping files for the Hibernate O/R mapper tool.

CODE WEAVING AspectJ, probably the most prominent example of code weaving is used in more and more projects. I have used it to „inject" log statements into normal Java code for debugging purposes in several projects.

Acknowledgements

First of all, I want to thank my EuroPLoP shepherd Arno Haase. He has provided many very useful comment on the paper, especially (but not only) with regards to structuring, categorization and overviews. Thanks!!

Second, I want thank all the pariticpants of the EuroPLoP 03 writer's workshop B, although some of their comments were rather harsh :-)

Several people have reviewed earlier versions of this paper and provided many useful comments. Ulrich Eisenecker provided a lot of very helpful comments regarding generative programming, product lines and frame processing. Alexander Schmid provided a load of useful comments and discussions on the topic, as well as on the patterns and their form. Last but not least, Uwe Zdun provided a wealth of very useful comments on language, style and of course, content and pattern writing. Thanks to all of you!

References

[Andrescu, 2001]   Andrei Alexandrescu, *Modern C++ Design*, Addison-Wesley 2001

[Apache, web]   Apache Group, *The Byte Code Engineering Library*, jakarta.apache.org/bcel/index.html

[AspectJ, web]   The AspectJ team, *Aspect J homepage*, www.aspectj.org

[b+m, web]   b+m AG, *Generator Framework*, www.architectureware

[Basset, 1996]   P. Basset. Framing Software Reuse: Lessons From the Real World.

Prentice Hall, 1996.

[BCEL projects, web]   Apache, *BCEL projects*, jakarta.apache.org/bcel/projects.html

[BOOST, web]   Boost group, *The BOOST C++ library*, www.boost.org/

[Bronstee, web]   Bronstee, *FUUT-je*, www.bronstee.com/spider/AboutFuutje.php

[Cleaveland, 2001]   Craig Cleaveland, *Program Generators with XML and Java*, Prentice-Hall, 2001

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

[d-s-t-g, web]   Delta Software Technology GmbH, *ANGIE*, www.d-s-t-g.de

[FI2010, web]   DoD, *Foundation Initiative 2010*, www.fi2010.org

[Gamma et. al. 1994]   Gamma, Helm, Johnson, Vlissides, *Design Patterns*, Addison-Wesley, 1994

[GMCL, web]   K. Czarnecki, *GMCL - A Case Study in Generative Programming*, www-ia.tu-ilmenau.de/~czarn/gmcl/

[Hyper/J, web]   IBM, Corp., *Hyper/J - Multi-Dimensional Separation of concerns for Java*, www.research.ibm.com/ hyperspace/HyperJ/HyperJ.htm

[ISIS, web]   Vanderbilt ISIS, *The Generic Modelling Environment*, www.isis.vanderbilt.edu/Projects/gme/default.html

[Kava, web]   Newcastle University, *Reflection Page*, www.cs.ncl.ac.uk/ old/research/dependability/reflection/

[Kennedy Carter, web]   Kennedy Carter, *iUML*, www.kc.com

[Kiczales et. al. 1991]   Kizcales, de Rivieres, Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991

[Koschmann, 1990]	Koschmann, *The Common Lisp Companion*, Wiley, 1990
[Metacase, web]	Metacase Consulting, *MetaEdit+*, www.metacase.com/
[Microsoft, web]	Microsoft Corp, *.NET Framework*, www.microsoft.com/net/
[Netron, web]	Netron, *Fusion*, www.netron.com/products/fusion/
[OMG,specs]	OMG, *Modelling and Metadata Specs page*, www.omg.org/ technology/documents/modeling_spec_catalog.htm
[OpenC++, web] openc++.html	Chiba, *OpenC++*, www.csg.is.titech.ac.jp/~chiba/
[Parnas, 1976]	D.L. Pamas, *On the Design of Program Families*, IEEE Transactions on Software Engineering, March, 1976
[Riehle, 2001]	Riehle et. al., *The Architecture of a UML VM,* www.riehle.org/ computer-science-research/2001/oopsla-2001.html
[Schmidt, web]	Doug Schmidt, *The ADAPTIVE Communications Environment,*www.cs.wustl.edu/~schmidt/ACE.html
[SmallComponents, web]	Voelter, *SmallComponents*, www.voelter.de/smallComponents
[Uni Karlruhe, web]	Univerity of Karlsruhe, *COMPOST/Recorder*, www.info.uni-karlsruhe.de/~compost/
[Voelter, slides]	Voelter, *Program Generation, A Survey of Techniques and Tools*, http://www.voelter.de/data/presentations/ ProgramGeneration.zip
[Voelter, web]	Voelter, Gaertner, *Jenerator - Generative Programming for Java*, www.voelter.de/data/pub/jeneratorPaper.pdf
[Vandervoorde & Josuttis, 2002] Gui	Vandervoorde, Josuttis, C++ Templates - The Complete Gui de, Addison-Wesley 2002
[Voelter, et. al., 2002]	Voelter, Schmid, Wolff, *Server Component Patterns - Component Infrastructures illustrated with EJB*, Wiley 2002
[XDoclet, web]	XDoclet Team, *XDoclet - Attribute Oriented Programming*, xdoclet.sourceforge.net
[XVCL, web]	Sourceforge, *XML Variant Configuration Language* fxvcl.sourceforge.net/