Theory of generation

tooling knowledge,

classification for generators

kinds or call 'em: pattern.

Boris-N. Wachowiak

Abstract

T. o. G., as abbreviated, a subject of an age, where programming generatively shifts into focus not only of researchers eyes. More practical solutions are known nowadays than few years ago. What's missing though still is a theory combining knowledge explored along with a bit of leadership on more approaches. anyway and with respect on leads:
let's go. & once again.

1. Introduction

..& to the material.

The overall subject „theory of generation" is not only because of it's non-existence one of the hardest for explanation. Difficult in general are the dozens of topics affecting the theory at it's heart. be it academic and research related, or everyday resulting engineering matters. Providing a framework is all but easy, on the other hand thoughts from others concluded own grown results at least to the point how important it is just doing it. here we are,

to highlight a major issue beforehand, the very mix of Software-Engineering will be the factor of success. at anytime.

the work in progress.

within here the focus is set on core characteristics framing technological aspects of generators as particular pieces of software. Jackson leads the way sorting the details like done before. Code generation as „a theory" best is seen as a particular frame of software problems consisting of sub-problems on their own. in consequence the frame provides a roof for the methods coming into question, especially when bridging the gap to solution issues, questions of design,

and of course: on ordering them in the right manner.

or more general: how, to.

like years have proven, better don't underestimate.

Different approaches are compared, discussed and assembled into further leading methodology. that way own ideas and principles, developed out of commercial projects, can be integrated, too.

subjects to care about can be as different as SW-Product-families, Parser-generators or Component-Technologies. G.P. means to be familiar with all of them. for sure, to some degree.

the goal as a whole is to further establish kind of an academic framework specialized on everything dealing with Generative Programming (abbr. GP) useful for professionals, academics and maybe

the both of them. to get it run:

the essentials.

2. Basic Generation Principles

.. some discussions.

when talking about code-generation, the selection and deeper study of works available has turned out to be tricky. stuff worth citing is not only difficult to select because of the huge amount available. Usually research on code-gen. tends to be highly specialized. thus making the evaluation of other people's work a tough one. Especially when the major goal is to have an approach that cares on the whole discipline. Promising works towards:
not too much I knew about.

2.1. Abstractions of Code Generation

.. reviews.

As the chapter's title gives the idea about, some general models are needed, to further classify G.P. Useful for understanding in the following at firstance shall be clearly distinctable structures known today thanks to others having brainstormed obviuosly a lot. Just for getting some progress, slightly extending them with respect to technological aspects playing important roles will be one of a few sub goals that can be addressed.

the approaches by themselves are not taken for their completeness. a major reason to leave them as abstract for now is the effect it has on further classifications later on.

Seen as a milestone, [01] gives a structure named „kinds of generators", that provide a useful classification heavily taken for this document's purpose. Herrington's work shows in detail inner pattern(in this case he is creating micro-architectures), all under the regime to construct practical solutions and explores the idioms used for the different types of generators he knows.

most probably my personal preference for the Herrington classification comes from the clear distinction they allow on generators anywhere.

An second approach of interest on classifying G.P. discusses similar questions(although different ones, too) and can be found in [02]. Comparisons are given wherever useful. The latter one is easiest described as being more academic, where the first focuses on daily solutions without forgetting some interesting theoretical pattern(even if not directly calling them such).

to get some details, let's have a closer look to the „kinds" and their major properties,

next page.

Table: Idioms, & kinds : Of Code Generation

although not dealing with real code anymore, doc-gen's, metric-reporting tk's and the like, it is common sense for academics to take them as code-gen's. hint: try to see the results as brain-code.

1. ***code munger*** GM

basic properties:
twisting and shaping code from one form into another
picks the important features to use them for output creation
usage examples:
documentation generators
indexing of important features, variables etc.
generation of software metrics

2. ***inline code expander*** ICE

properties:
takes some source code as input to create production code
the latter is hidden to the developer
input code contains special markups
usage examples:
Embedded SQL
C/C++ Macro Processing
else:
simplifies the code by adding specialized syntax, difficult to debug

just a personal opinion on numbers 2 and 3. ready made: worth using. doing yourself: try to avoid.

3. ***mixed code generator*** MCG

properties:
reads a source file, modifies and replaces it, looks for specially formatted comments, creates fragments of production code for them,
markup principle once again. uses are similar like for inline code expansion

usually the best kind to lean on when having to create some generator at any point in time.

4. ***partial class generator*** PCG

properties:
uses abstract definition files as input to build set(s) of classes
output creation most probably done using templates
hand crafted code added for compilation time, to complete the production set
a starter to create tier-generators
usage:
whenever looking promising

besides those skipping with frameworks sometimes; whenever you've done manually: you're way down the road. immerhin.

5. ***full-tier / layer generator*** FTG

builds complete layers of n-tier systems, no custom or derived classes
structure or kind of input type very much the same than for PCG's
templates for output creation
usage examples:
database-access layers and the like
else:
desing for special cases can be problematic
commonly build out of PCG's

famous for being the most effective kind of generation principle, but a rather broad topic on it's own. check [Dom-Proc] or dozens of others. usually really: specialized.

6. ***domain generator*** Dom.-gen.

definition file(input) is an instance of a full domain language,
touring complete, to allow the representation of comain-concepts more easily
advantage:
high level description of solutions, very close to requirements
disadvantage:
training for new language necessary,
complex to build, support for documentation, testing etc. is lacking
else:
can be build out of FTG's

generators of the kind occur e.g. when dealing with EJB's. Similar to usual documentation generators, source code is taken for (multiple-) output production whether it be code again.
a widely used of it's kind: the xdoclet OS-toolkit, stating by name enough to see the idea behind. the major reason for highlighting like that & the simple fact, how sources carrying such tool-based meta information used to generate more code, by themselves can be results of generation-runs; model-gen's for example, but others, too.
having that it's rather trivial to conclude pipelining as one of the most important, as well as interesting, sub-problems within ideas on overall code generation.
we will have to remind of.

2.1. ..in continuation.

As the abstract gives an idea about, Voelter mostly roots to MDA, itself being an explanation for his concentration on terms well-known from modern MG approaches, where [Herr03] wants to be more general.

so much on kinds of generators and for the time being. [2] as mentioned, cares about similar questions, partly, because Voelter is heading a bit different, like mentioned as well. Anyway his work gives reason enough to cite a bit detailed, and be it for the overlappings in comparison to [01], that sometimes are hiding.

A popular means for a couple of years already, the term of pattern found it's way into any specialized form, and this time a collection of Patterns for Generation, provides another overview what has to be known about generators in case of plans to follow that road. let's take a mere view of seven patterns named, occuring when studying the inner life of generators. the listing again:

Table: Code-Gen. Patterns

Templates + Filtering	Templates anyway for sure are a technique of high importance for engineers being able to apply.
production code is embedded into text templates; some textual specification(e.g. XML) given for input; relevant information to be filtered out.	
Templates + Metamodel	collecting patterns like that, I'd suggest [Bass 97] more as "Templates+Filtering", because that's what he does.
extension of the first one; process of code production controlled in terms of domain concepts	
Frame Processing	
characterized as „functions that generate code as result of their evaluation".	
API-Based Generators	don't understand what's meant by now.
„provide an API against which code-generating programs are written. this API is typically based on the metamodel of the target language"	
Inline Code Generation	Code Weaving got popular as a following of the restrictions Java had in comparance to C, C++ or others. that's why C.W. is IDE at it's heart. or written different: modernized code preprocessing. however.
the same like the „Inline Code Expansion" kind.	
Code Attributes	
attributes within „normal" source code(typcially annotations) are filtered out to produce more code; similar to „Code Munging"	
Code Weaving	
weaving different parts of program text together. think about AspectJ.	

but what? can we learn by that.

2.2. Techniques .. what's worth using.

When talking about industrial like code production, it's hard to avoid taking some time for the discussion on methods applicable. Code Generation itself fits into a particular frame of software problems. Certain techniques, tooling-kinds or more general: knowledge areas that can be identified for. The hard job is the big collection of sub-problems a frame like this one consists of. The last chapter gave some hints but let's move forward caring about skills needed, like called in [01] once again. summarizing his collection looks as follows:

<n.p.>

Table: Code Generation Skills

	skill naming / short explanation (cited)	
1.	***Using text templates*** „Generating code programtically means building complex structured text files en masse." „Using a text template means that you can keep the formatting of the code separate from the logic ..."	that's what it's about. although it's still not everything.
2.	***Writing regular expressions*** „Many people learn to use regular expressions as invocations of black magic; they use them without knowing precisely why they work. Regular Expressions are an invaluable tool and worth your time to learn thoroughly. Once you have mastered them, the idea of writing string parsing code by hand will be forever dashed from your mind."	it's: black magic. but that's why he's right.
3.	***Parsing XML*** „XML is an ideal format for configuration and abstract definition files. In addition, using schema or DTD validation .. can save you from building elaborate error handling into file-reading code."	it's a starter. working more. coding XML: too messy.
4.	***File and directory handling*** „Code generators do a lot of file reading and writing as well as directory handling." „The most common scripting languages have built-in, easy-to-use APIs for directory construction, directory traversal, and pathname munging, which work on any operating system without code alteration or special cases. The same is also true of file I/O, which is implemented in the core libraries of these languages."	? the main reason to use, when programming gen's. performance usually isn't an issue then.
5.	***Command line handling*** „Code generators are usually command-line based. They are invoked either directly, as part of a check-in or build process, or from an IDE."	they are more: kind of files. keep in mind

Besides everything that can(and will) be critized, the technological buzzwords listed are a pretty fair idea on what developers need for building a toolbox or working generatively. on the other hand, they are most likely not complete. anyway the re-listing will be postponed for later on.

for sure the listing came after years of thoughts how to extend.

As Voelter shows, too, text templating tools and languages can be seen as one of the most frequently required skill when programming generators. Closer looks on tools available on the web show quickly that they are a very popular means nowadays(and not only for code-gen). Most likely they are, because that way repeating code fragments can be generalized and maintained pretty easy to some degree.

having Bassett's SCV-Reuse levels in mind. [Bass 97]. around level two.

Another skill of high importance: regular expressions. Useful anyway when building software systems, it's one of the secrets for modern software engineers and again:

worth some notes on their own.

not only those working on G.P.

for detailed reasons, [08] shows why MG is multiple stage processing, here in terms of Input- & Output-models, or why they are better kept apart.

Having MG's further fulfill goals of all sex kinds is none was an 'idea', born to [MDA exp.], and resulting brain-storming on the PIM, PSM to code-cycle. [Zachman] from my point of view just would help control the architecture part of MDA for what it should be: architecture.

OCL ?! still kind of a mysterium. a fascinating.

One of those works specializing on OCL, [08] for example, show easily, it would be something, but nothing more, than tricky coding to implement and extend the ideas of Warmer, Kleppe and Bast in sense of generative methodolody; and just so much on what could be done today. The discussion was about 'mystic types', and therefore back to the pioneers of the Object Constraint Language. Themselves, beyond everything, show some facts that are of major interest herein especially for collecting more arguments on the idea of „the seventh kind". The kind of transformations [04] provide us with are inline code expansion at their core, sometimes mixed code generation can be done on a model, in case the developer should be able to re-edit some expanded model manually. Besides ICE-generation, which can happen, Mod-Gen's further fulfill goals of PCG's, FTG's and why not Dom-Gen's. For capturing the idea of model driven code production the focus best is set on the word model as such. By that a different view further argues a separate „kind". In consequence the point of investigation is nothing but the language used to specify and describe some model.

To „instruct" the computer what shall be done, formalisms are still unavoidable. Modeling languages, UML is the most popular example for, are semi-formal. Languages, as simple as might be, used as input for the other kinds are formal in the sense that the generator carries the meaning of the languages constructs. In case of modeling languages you always have to do some work to make the generation software „understand" the models meaning. expressed slightly different, that's what MG is about, creating useable libraries that combine specifications and configurations to describe a model's semantics. UML for example again, enables the software engineer to define such configurations, called „UML-Profiles", collections of Stereotypes(defining a particular understanding of a concrete model element), Constraints(most probably written in OCL). model generation software needs to enable the developer to make use of profiles while supporting him with the specification of code abstractions related to the Profile's Definitions. Practically spoken, in case of semi-formal languages, more has to be done than if you're using some custom language. Within profiles elements of interest for generation are declared, just the definition of the resulting Code-Structures is specified in separate and on top of them.

To summarize model driven generation for the time being, few statements can be made here. Lanuages overall play a very important role when classifying Problems or Patterns of Code Generation. ongoing research especially on MDA, better expects OCL as a big secret for understanding the whole „kind" itself. The problem with OCL is similar to what's written above on Modeling Languages: The applications are unforeseen, but for getting the right tools, you may have to build them from scratch.

more thoughts on languages are to follow.

well. what's needed technically, of course important to agree on. anyway. some minor detail comes as a bug weakening the approach. the grand mistake.

Some deeper investigation on Herrington's summary follows it's time and will allow conclusions to broader classifications. e.g. parser generation, combining particular of his skills into a well-known research area itself, and leaving some space for expansion that way.

Expressed the diplomatic way, beyond all that, influencing the complete set again, some techniques play bigger, not to say: major roles, and others are still missing. As for practical purposes, although developing a theory, the end-product is nothing but some piece of software, typical problem-solving approaches need to be included additionally.

just let's do that starting from Chapter 3. there's still

a hole in the bucket, o' Henry.

2.3. The 'mystic' Type

..the unknown kind.

the leading question when arguing MG's as a seventh kind is the one dealing with: "what makes them different" to take one before the details, closer thoughts make pretty obvious, that, in one way or the other, they provide a root for the sex "known already".

In Section 2.1 remarked, right now is the time and place to argue the six kinds to be truly seven of them. which one, what reasons.

Although Herrington is not exactly wrong by taking model generators as full-tier generators(the generation of whatever tier can be done anyway), he makes a very important mistake, explained while highlighting some minor items dividing them.

First of all, already mentioned, full-tier generators tend to be bound to one(or few) framework(s), either own or third-party, the generator produces user-code for. model based generation on the opposite does nothing say about frameworks selected for application. To keep it simple at this point, MG's can generate against any such framework, api etc., and which could be thought about. to go a bit further: MG's have a completely different, more broader, scope than FTG's.

Later on additional arguments follow, why and how MG's are one meta-level higher than others(summarized: for language reasons). practically it means so far, that what you have to deal with, is the problem of enabling a generation software to produce against any framework of your choice. Back to FTGs, it's pretty obvious that they carry this capability already within them, tiers usually highly build on particular frameworks, „framing" the tier. for model based generation frameworks are not the direct scope.

Ideas about Model Driven Architecture(abbr. MDA) [04], show pretty clear, how much potential still is to be explored. e.g. the usages of OCL thought as a textual part for the specification of constraints on a model, is hardly to be seen completely today. For example the language is deisgned pretty well, too, to specify queries against particular models, seen in [9], or anything else, e.g. [10].

personal viewpoints assume domain based generators as the kind to prefer over their counterparts. throughout history programming computers was at it's best, when done the textual way.
Graphical and similar types of programming have very few uses overall. remember [cf. F. SA.] to remind argumentation links. finding more reasons for a preference like above is a hard task but it's just a preference anyway.
besides all of them that could be argued, most probably touring-completeness can be suggested, as well as the knowledge how to refactor towards. and just so much on personal major ones.
anyway again the term as such still is hard to get. a systematic approach on building them: hardly to be seen. model-normalization should be a success factor for the future.
domain modeling and language design, the tasks to be done mainly should become
a lot easier then.

3. Classifications

..to get the Edea.

Having focused on what has to be kept in mind doing G.P. professionally, moving on slowly automatically leads to some reorderings. For chapter 3 the main goal shall be sort of a meta model for G.P.-tooling. Jackson rules.

3.1. Characteristics

..avg.-case: I/O-ruling.

So far we haven't done much else, than discussing some foundations, critized what others seem to have done wrong and promised to do a bit different. Heading there it should be worth mentioning, that besides all criticism one thing must never be forgotton. Code Generation as a discipline is still in it's early ages. The first compendium can be seen in [05]. Anyway do works like Czarnecki and Eisenecker's or the highly referenced from Voelter show, generation of sources goes back into years of C/C++-Preprocessors or the like, only a „complete" discipline lacks clear principles until our very days.

on the hunt, issues, for more design.

The „kinds", discussed in detail, mostly concentrate on available inner processing concepts; what frames the generator when seen as a particular type of machine. Beyond that and all to be kept in mind on model generation, the known kinds clearly can be divided by certain characteristics, leading to abstract distinctions establishing a class-scheme for all of them. let's discuss what can be seen easily, sort of properties, putting things together.

First of all, the number one property, the Files processed, further divided into input and output. Kinds 1 to 3 have in common their input being written in typical programming languages like C/C++, Java and lots of others. The special concepts, markups, usually are embedded into the language therefore slightly extending the language as such. Kinds 4 to 6, with respect to input operate on custom languages optimized for particular solution domains.

„Kind #7", model generators, have neither input like that. they have to do process abstract system descriptions defined using some modeling language, which are very different to the input types for 1 through 6. Model code neither is a language for writing executable programs(with few exceptions), nor is it a custom or „highly specialized" one.

Property number two, the output files created by the generator. Back to #'s 1,2,3, the output can be program code again, typical for 2 and 3, but not necessarily is, especially in case of CM's, remind Section 2.1. For all the three of them it can be said, that the output could be of any type, text, hypertext, or even binaries could be thought about. 4,5, and 6 always produce real source code for whatever language imagined. two classifying terms can be derived, before leading to further characteristics immediately then.

Concluding the circumstances of In- and Output-Types, it's possible to clearly distinct between two classes of generators. 1, 2 and 3 generalize as „extending-", 4,5 and 6 as „transforming-" types of generators; #7(Mod-Gen's) best is typified as a mix, building a sum of both's overall properties, but we found them out to be different anyway. Model Generators are Type „Extending", because they extend the code base as such, and they are „Transforming" due to the fact that a different representation of an abstract definition is finally mapped, or: transformed, into a touring-complete description, most likely some usual programming language again.

The overall metamodel prototype can be seen in Figure 1, sort of a summary for the discussion of the page before, and giving an overview of properties discussed.

the graphical way.

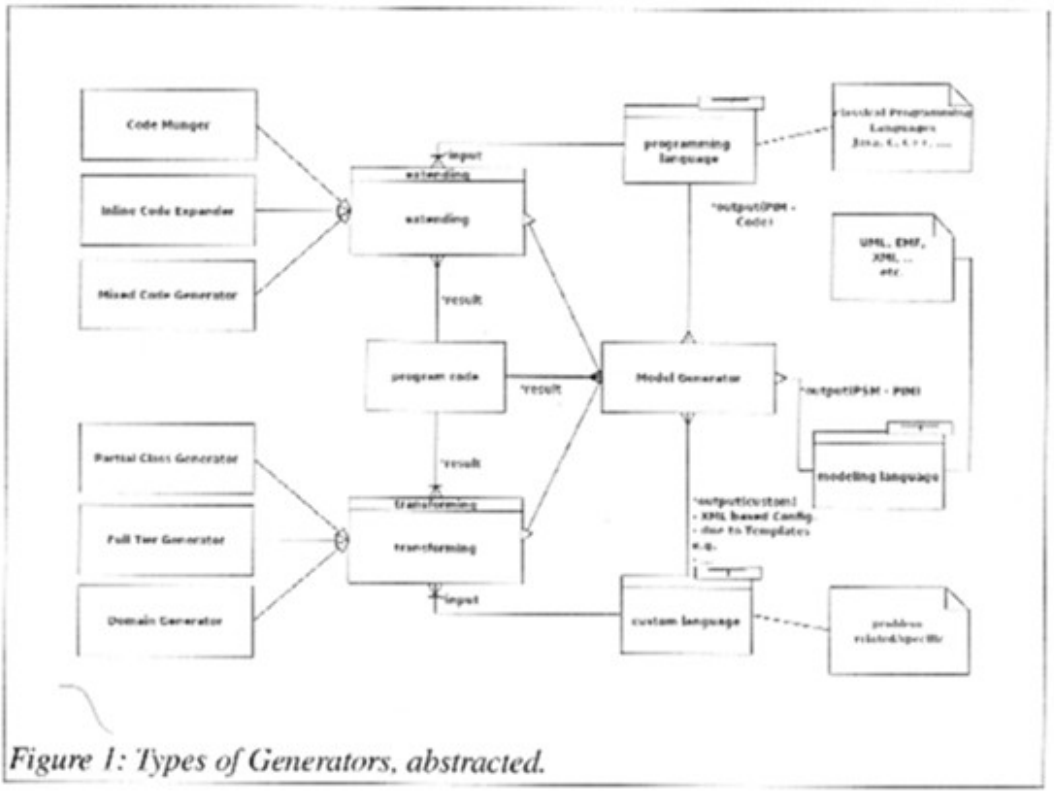

Figure 1: Types of Generators, abstracted.

to present a possible misunderstanding, Herrington's 'scheme' was the one giving an idea on "T.o.C." for the first time. that's what it turned out to be, by now. better: the decision how to follow him.

.. characteristics concluded

Having established distinctions like that, we can move on a bit now. taking a side step, especially on the input aspects leads next to questions of interrelations between the kinds, more general the newly found types. practically spoken, in which way do the different generators provide input to each other. for sure another question of major interest for tooling, the one we always keep in mind. Predicting particular flows of generation within particular projects, somehow it's kind of impossible, although simple laws can be followed out of the Types discussion.

Starting with M-Gen's this time, it's no big deal to imagine their results as possibly being input to all the other kinds. Generally spoken their results can be all type of source language, be it real code further being processed with by Code Mungers, e.g. remind the xdoclet-toolkit. or be it code written in any custom kind of language. XML-Configuration files jump into mind, very typically used for configuration problems by dozens of availabe software packages today; maybe they come along as a result of a model driven generation run. Themselves, such XML-Config's, again could be used on their own as input for generator programs shipped with SW-Packages however you imagine. More general, results of transforming types(the respective „kinds") can be input to expanding, always under the regime of some complex generation pipeline flow. The most simple example to mention once again: Documentation Generators. Within professional generation flows, you most probably will produce documentation of generated code, too. The other way around, results from expanding have kind of end stadium. no more generation should be expected from here on.

so much on that, for the time being, but leading to deeper conclusions from the methods point of view. look out for Figure 2 thought to explain the „laws of generation flow" derived above.

The overall metamodel prototype can be seen in Figure 1, sort of a summary for the discussion of the page before, and giving an overview of properties discussed.

the graphical way.

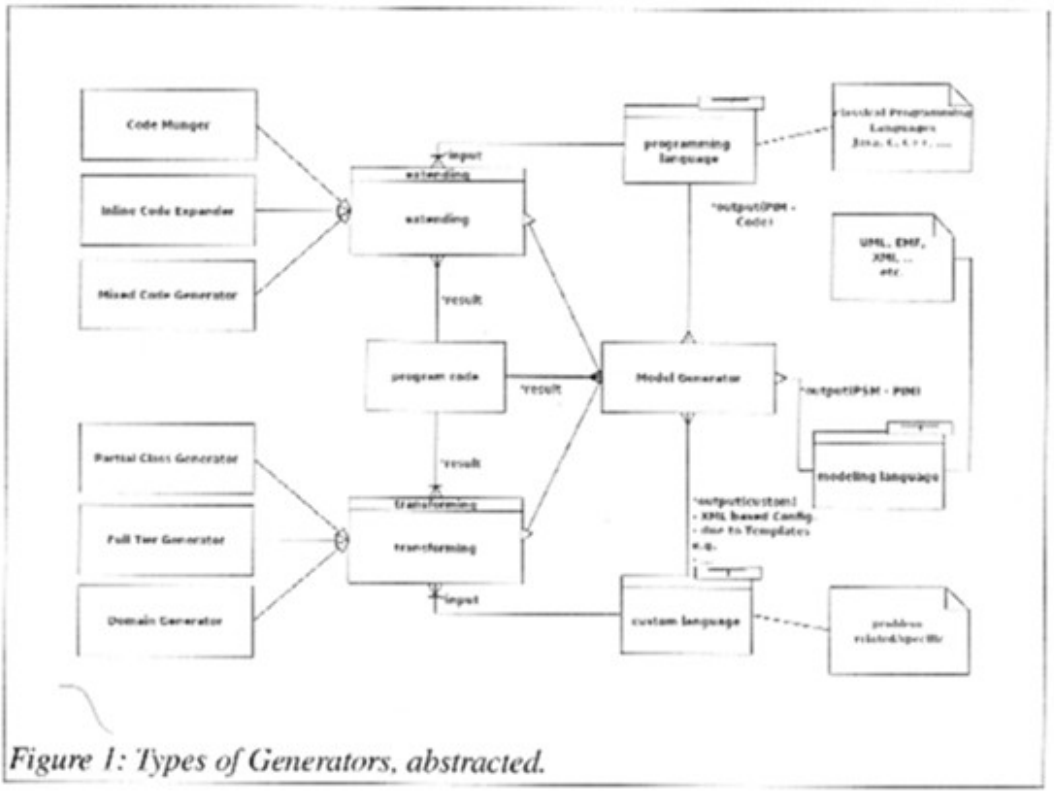

Figure 1: Types of Generators, abstracted.

to present a possible misunderstanding, Herrington's 'scheme' was the one giving an idea on "T.o.C." for the first time. that's what it turned out to be, by now. better: the decision how to follow him.

.. characteristics concluded

Having established distinctions like that, we can move on a bit now. taking a side step, especially on the input aspects leads next to questions of interrelations between the kinds, more general the newly found types. practically spoken, in which way do the different generators provide input to each other. for sure another question of major interest for tooling, the one we always keep in mind. Predicting particular flows of generation within particular projects, somehow it's kind of impossible, although simple laws can be followed out of the Types discussion.

Starting with M-Gen's this time, it's no big deal to imagine their results as possibly being input to all the other kinds. Generally spoken their results can be all type of source language, be it real code further being processed with by Code Mungers, e.g. remind the xdoclet-toolkit. or be it code written in any custom kind of language. XML-Configuration files jump into mind, very typically used for configuration problems by dozens of availabe software packages today; maybe they come along as a result of a model driven generation run. Themselves, such XML-Config's, again could be used on their own as input for generator programs shipped with SW-Packages however you imagine. More general, results of transforming types(the respective „kinds") can be input to expanding, always under the regime of some complex generation pipeline flow. The most simple example to mention once again: Documentation Generators. Within professional generation flows, you most probably will produce documentation of generated code, too. The other way around, results from expanding have kind of end stadium. no more generation should be expected from here on.

so much on that, for the time being, but leading to deeper conclusions from the methods point of view. look out for Figure 2 thought to explain the „laws of generation flow" derived above.

on Templates

Above all criticism on too much using them a couple of things never shall be forgotten.
(Text-) Templating languages are themselves full-fledged programming languages. With respect to 'lands' they are touring-complete, making them pure domain-languages for the problem of producing structured fragments of Text.
In terms of Types' Templating as method comes into play for transforming generation. Expanding lands on the opposite mainly rely on parser-technology, applicable on Dom.-len. besides Templates, too. Abstracted to model-driven this technique has been proven to be effective for the production part within the transformation flow. For semantic analysis of modeling-data, well, a bit & it depends on languages used, but 'real' OCL-integration might stabilize level 2 for Templating-approaches. Just don't expect levels 3 or 4, so soon. [Bass 97] takes it as one of his means, but optimizes towards flow-processing.

3.3. techniques revisited

..and to extend them.

Right on, Sections 3.1 and 3.2 gave an idea of methodological aspects necessary to provide some kind of cookbook. 3.1. further classified generators as pieces of software themselves, heading to particular, clearly distinctable problem frames of interest, discussed in detail in 3.1. In the end it is as simple as G.P. needs some means to structure the doing aspects of the programming principle as such. Besides the frames discussed in previous chapters still more has to be done.

In terms of an overall structure for the generation problem itself, methods of any importance need to be framed, extended and newly structured. Beyond skills defined by [1] and the analysis out of Section 3.1., finally a re-definition of Herrington's collection is easy to derive,

no big deal, and once again the summary:

Table: Skills Revisited

	skill naming / abstract.	
1.	***templating*** VTL, JSP, and lots more.	and not only because of the model driven approach, of course: important.
2.	***parsing technologies*** XML, regular expressions, grammar based parser programming, e.g. Yacc, JavaCC, Flex, ... ,AntLR	maybe the most underestimated area of knowledge within the SW-business.
3.	***file- / directory- and pipeline - handling*** IO Programming, API-based, Command-Line Driven.	you better know the roots.
4.	***scripting*** Ruby, Perl, Javascript, Bash-Scripting	and re-learn how simple it can be.
5.	***patterns and framework building*** patterns of all kind: design, architectural, Idioms. Pattern-Formalization, API-Programming and Design, remember the apache-story.	good programmers: do it anyway.
6.	***problem oriented analyzation and domain engineering*** Problem Frames, FODA	always start programming in mind. then it can be oh so easy
7.	***meta modeling***	and never forget: patterns.

remarked on the listing above, guess it. but that's what you need for really doing G.P., especially in Praxis and technically spoken. seen from a meta-level it's meant as a scheme as well useful to include missing examples left out for reasons of discussions they'd raise. there's more.

Side-Step:

the summary how gen's interrelate as part of pepeline-flows. kind of a secret to understand the whole thing.

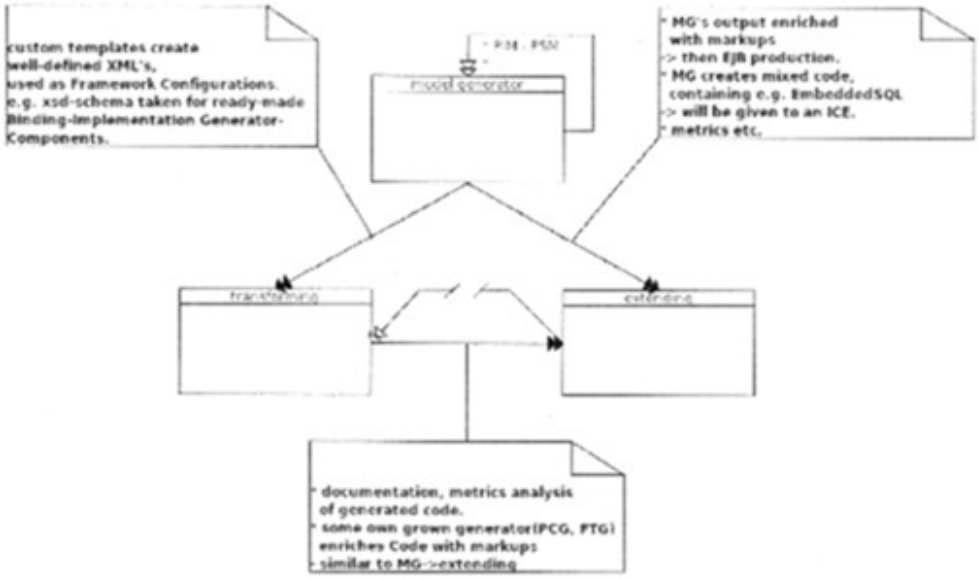

Figure 2: Generators Input-Flow to each other

Not to be forgotton as well and to keep on completing the cookbook on G.P. some rules programmers always have to keep in mind. The ten as such, like their author describes it: „a handy set of rules that you can use when you are designing, developing, deploying, and maintaining your code generator":

Listing: Top Ten Code-Generation Rules

01. give the proper respect to hand-coding

and not only for those of you programmers. what else.

02. handwrite the code first

but optimize. refactor.

03. control the source code

as well as an intelligent such.

04. make a considered decision about the implementation language

better believe him.

05. integrate the generator into the development process

although you will like it.

06. include warnings

.dito.

07. make it friendly

& you even might hate it.

08. include documentation

people are stubborn and there's nothing you can do about. like any other software you have to keep the code clean.

09. keep in mind that generation is a cultural issue

10. maintain the generator

4. G.P. beyond ..& for the future.

final chapter, before summarizing, concluding sections. Founding on numbers 2 and 3 it's time to browse issues left, followed by best case scenario(s) hardly to be seen. Worth a bit reflection (Sect 4.1.) the fact of languages; one of the most tricky overall, requirements tend to hide a bit.
Having strived a term as complex like, all that's missing is a mere overview where future could end up, but not necessarily will (4.2.). Just wow, move on.

4.1. languages ..an abstract.

Difficult, but the best choice to concentrate on knowledge areas, it's a good abstraction to put issues under a certain roof once more. The big deal within the G.P. paradigm is the amount of computer languages that have to be cared about, that you wanna have integrated, while their special features all have to be given the proper respect. And it comes even harder, you have to do everything on two levels. Construction-Time, Run-Time.

die Binsenweisheit

If only one fact is written by now, the T.o.G. is one particular frame of software problems. Following from the circumstances that some languages are more related to generator code itself, while others only relate to the solution's code. It's hard to predict like so much else. some of them can be formal, semi-formal etc. In case you start in the typical manner, building a PCG, it is the generator implementing the formalism you mean for the language's elements you constructed, and it's similar for the other transforming types. For model generators, the whole story is a bit different, you get formalisms by the definition you create how to interpret the models element; you have to do it in one way or the other. Look into [6], how, it shall be done. And then, even expect quasi-formal languages, for example taking an excel sheet as kind of abstract definition used for input to a generator component.

On formalisms: it part of the IT-business for some time. it should be easy to imagine, that without formalization you'll never be able to instruct computers to your needs.

Always discussed as the most useful approach of generation: domain languages. [01] does not care about them any deeper, it is a rather broad topic with unkown amounts of research works. Check [03] for interesting facts about and for some ideas what has to be done, with respect on domain-based code generation. Additionally,

Mentioned earlier already as one of the main secrets behind G.P.. Having used them once should help getting the respect necessary for doing twice and see the benefits about. Whenever done a couple of times, it is nothing but fascinating in which ways they help getting control over formalization of strings. generators of all kind deal with languages on both ends of the pipeline. So what else could it be, you want as a means processing compact micro-languages jumping out of your everyday developing mind?!
useful for 'normal' SW-production as well, formalization of file- & directory-processing is just a jump over the fence, once, you got the magic behind.

Unix-Administrators swear on them, they should have reason for. expressed the other way around one more time: they're ready to use.

a very useful discussion on the term can be found in [Jack95]

and really a difficulty overall, the term of „a domain" sometimes lacks of a clear understanding, further confusing the audience. Important about, mentioned as well already, such languages are touring-complete by definition, making them extremely popular, not to say: highly effective. The concepts are hidden in the generator, just building it, a different story. Meta-modeling probably could be one of the means. and with respect to hints: better try getting there.

4.2. overall code production

..another idea. finalizer

right now having collected not all but quite a lot to be known for doing G.P. professionally, the road's end to full blown approaches, and generate code as part of money worth projects is a completely different story. giving an outlook for „O.C.P." is worthwile anyway.

It's not the objective within here to tell: „the title will be reality at point in time x or y. One of a few subgoals was the discussion of what is required from a technical point of view when doing though. The list of tools to use, methods to apply, although the amount looks huge(remember 3.3 and the predecessor) is an endful one.

that's what I think about. for quite some time. already.

for what it should look like, O.C.P., Figure 3 is an indermediate conclusion. and to be honest, it's a completion in parallel to all the text about. not only for titles like 'this'.

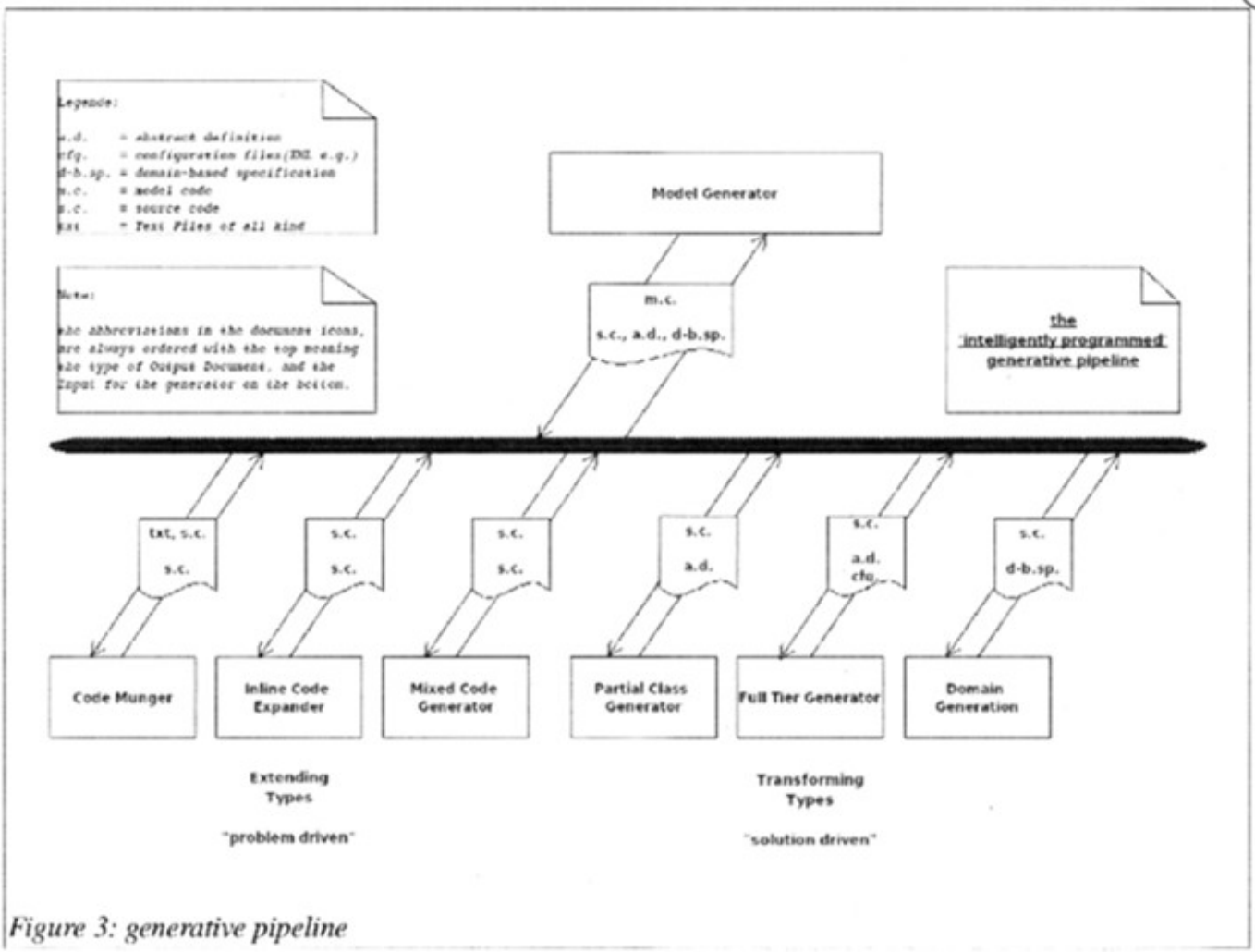

Figure 3: generative pipeline

Summary

well then. within this very document some mountains to climb for a Theory of Generation finally got summarized and road-fragments could be concluded by that. The big reference to highlight, for reasons of Icon-respect: Problem Frames. without: expect to get lost, and with them: expect a long term learning curve.

C.P., of self a paradigm in parallel to others, like OOP, scopes professional mass production of code in a systematic manner. It does not mean 'a single programmer uses one, but management doesn't know about'. Just this one's a challenge on the project managers'. rounding up, let's look over the wall.

Still more time lies ahead, than behind, and not only with respect to the "C.f.G"'s predecessor. and as much are there "questions" not discussable anymore for space and complexity reasons. e.g. "C.P. and overall project culture" or "C.P. for team-management", to take just two of them.

It is a complex problem and within here nothing but few hints for getting it under control have been collected. for completion's purpose:

that was the idea about.

B. Bibliography

abbr.	abbr. scripted	Author / Title / Publisher / Year.
[01]	[Herr 03]	Herrington, Jack. Code Generation in Action. Greenwich, CT: Manning, 2003.
[02]		Voelter, Markus. A Catalog of Patterns for Program Generation. EuroPloP2003, Eighth European Conference on Pattern Languages of Programs, 25th--29th June 2003, Irsee, Germany, http://hillside.net/europlop/papers/WorkshopB/VoelterM_1.pdf, 2003.
[03]		Cleaveland, J. Craig. Program Generators with XML and Java. The Charles F. Goldfarb series on open information management. Upper Saddle River, NJ: Prentice Hall PTR, 2001.
[04]	[MDA exp.]	Kleppe, Jos; Warmer, Anneke; Bast, Wim. MDA explained – The Model Drive Architecture: Practice and Promise. The Addison-Wesley object technology series. Boston: Addison-Wesley, 2003.
[05]		Czarnecki, Krzysztof; Eisenecker, Ulrich W. Generative Programming - Methods, Tools, and Applications. Boston: Addison Wesley, 2000.
[06]		Fontoura, Markus; Pree, Wolfgang; Rumpe, Bernhard. The UML Profile for Framework Architectures. Object Technology Series. Addison-Wesley, 2002.
[07]		Jackson, Michael A. Problem Frames - Analyzing and structuring software development problems. Harlow, England: Addison-Wesley/ACM Press, 2001.
[08]	[08]	Akehurst, David H.; Patrascoiu, Octavian. OCL 2.0 - Implementing the Standard for Multiple Metamodels Electr. Notes Theor. Comput. Sci. 102: 21-41, 2004.
[09]		Akehurst, David H.; Bordbar, B. On Querying UML data models with OCL. Lecture Notes in Computer Science. no. 2185: 91-103, 2001.
[10]		Giese, Martin; Hähnle, Reiner; Larsson, Daniel. Rule-Based Simplification of OCL Constraints. 2004.
[11]	[Bass 97]	Bassett, Paul. Framing software reuse – Lessons from the real world. Yourdon Press computing series. Upper Saddle River, NJ: Yourdon Press, 1997.
[12]		Kovitz, Benjamin. Practical Software Requirements: Manual of Content and Style. Greenwich, Conn: Manning, 1999.
[13]	[Dom.-Proc]	Lengauer, Christian; Batory, Don; Consel, Charles; Odersky, Martin. Domain-Specific Program Generation – International Seminar Dagstuhl Castle, Germany, March 2003. Revised Papers. Springer Verlag, Heidelberg, 2003.
[14]	[Jack 95]	Jackson, Michael A. Software Requirements & Specifications: A Lexicon of Practice, Principles, and Prejudices. New York: ACM Press, 1995.
[15]	[Zachman]	the works and ideas of John Zachman. check: www.zifa.org

B. ..completed.

[16] [f.F.SA.] b.n.w.
Theory of Generation – some ideas.
The Zachman Framework for the reporting problem. a first approach.
sent out by mail „Zachman Framework for foXf-Architecture", to everybody.
July 2006.

I. Index of Tables, Illustrations and Handcraft.

Tables

Figures

Script Highlights

The golden Rule

„handcode first" :

exactly.
but do the right way.