Theory of generation

.. Some ideas ..

The Zachman Framework for the reporting problem

- a first approach -

Preface

Within this document an overview is given for a framework, implementing a theory of generation. It is beyond it's scope, to design a framework in the classical manner. Instead, a loose collection of ideas is presented, that grew over a certain period in time, with always having the 'best-possible' generation toolkit in mind.

Subject of generation in principle can be all kind of documents. By the application of the Zachman Framework for software architecture, the most important issues, playing a role for generator theory, will be presented and discussed a bit.

As the idea for a theory of generation needs a simple approach for an overall method dealing with the documents to generate, a pragmatic decision is unavoidable. The Zachman Framework is the best known principle defining such an approach with respect to the author's knowledge.

To undermine the ideas explained here, the foxf software for report programming serves as an example, and is used to bridge the gap from academic background information to industrial reality.

For the application of the Zachman Framework with respect to generators, the next chapter gives an overview of the ideas behind. Last but not least, always keep Problem Frames in mind, Jackson rules.

to undermine the usually very abstract ideas, a case study helps keeping the path for a discussion of important terms and concepts. all under the topic of generation.

The Zachman Framework

The Zachman framework for software architecture (abbr. ZF) helps organizing all kind of Software artefacts within a table like structure, consisting of columns and rows and therefore building particular cells. Each cell within that framework carries certain kinds of information with respect to some or the other formalism, meaning particular languages, textual and/or graphical. A very broad and mostly complete overview of languages can be found in [Hay 2002].

The application of this framework as found within here is a mere demonstration towards an implementation of Zachmans ideas using UML. The ZF rows and columns represent viewpoints and issues framing the content of a software's collection of documents.

To avoid confusion, some remarks regarding the scope of this document have to be made. This very model just names models, to be created for the documentation of the foxf-software, and/or names the kind of artefacts to be found for the different rows and columns.

the main idea behind the overall model:

- get a means that helps structuring the kinds and classes of documents found within particular software projects.

as of this, the subject used for discussions is the problem of programming printable reports.
It is taken as a starter to get a generalization for documents to be generated.

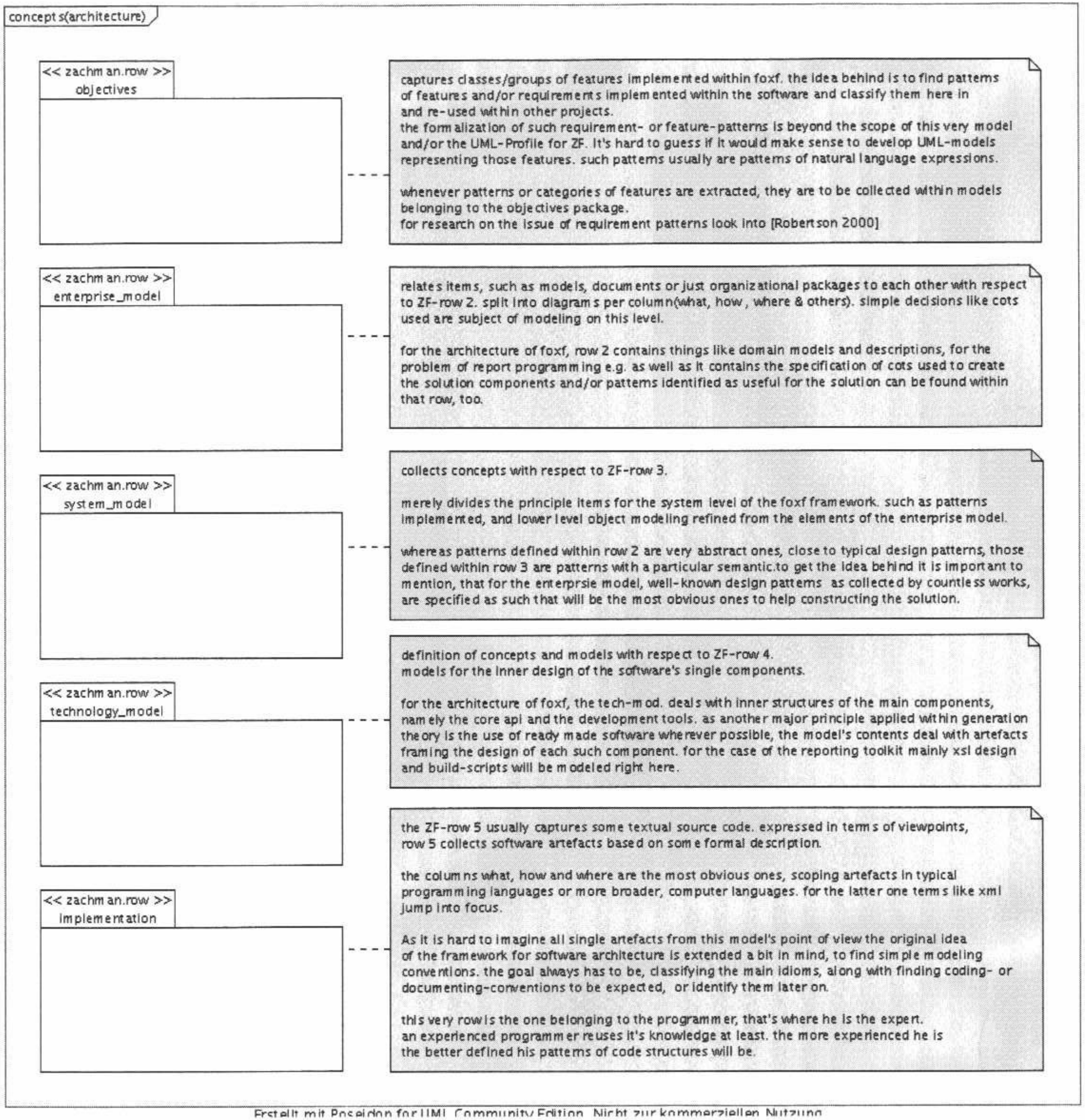
main scopes for academic purposes are the following:

- simplify the re-use of all kinds of software artefacts
- find further classifications and patterns for all those documents

To start working towards an UML Profile for ZF, simple stereotypes are established by time and in the future. The idea behind those stereotypes as understood under the term of generation-theory is, to get a library developed for the goal of getting some generators for development-tooling code, not to confuse by production code.

Expressed a bit more practically, the build-scripts necessary for each project are subject of generation themselves. Further build-time software artefacts such as configuration files for metric tools like dependometer, are within the scope of such a profile too.
It can be argued quickly, that as the UML-Profile for ZF has to care about major artefacts only. Their type-modeling and classification is the task to do. models using that very profile could be titled 'project configuration models'. pre-generation of models to be developed, build-time code and others, should be a more realistic problem to solve then, having such a profile.
On the other hand Zachman's ideas don't mean a framework under the usual term's definition. Just having his ideas in mind helps controlling the expected complexity of overall generation.

deeper investigations of UML-profiles would be beyond our scope again.
although,
some alternative ideas about their usage may be worth mentioning.



captures classes/groups of features implemented within foxf. the idea behind is to find patterns of features and/or requirements implemented within the software and classify them here in and re-used within other projects.
the formalization of such requirement- or feature-patterns is beyond the scope of this very model and/or the UML-Profile for ZF. It's hard to guess if it would make sense to develop UML-models representing those features. such patterns usually are patterns of natural language expressions.

whenever patterns or categories of features are extracted, they are to be collected within models belonging to the objectives package.
for research on the issue of requirement patterns look into [Robertson 2000]

relates items, such as models, documents or just organizational packages to each other with respect to ZF-row 2. split into diagrams per column(what, how , where & others). simple decisions like cots used are subject of modeling on this level.

for the architecture of foxf, row 2 contains things like domain models and descriptions, for the problem of report programming e.g. as well as it contains the specification of cots used to create the solution components and/or patterns identified as useful for the solution can be found within that row, too.

collects concepts with respect to ZF-row 3.

merely divides the principle items for the system level of the foxf framework. such as patterns implemented, and lower level object modeling refined from the elements of the enterprise model.

whereas patterns defined within row 2 are very abstract ones, close to typical design patterns, those defined within row 3 are patterns with a particular semantic.to get the idea behind it is important to mention, that for the enterprsie model, well-known design patterns as collected by countless works, are specified as such that will be the most obvious ones to help constructing the solution.

definition of concepts and models with respect to ZF-row 4.
models for the inner design of the software's single components.

for the architecture of foxf, the tech-mod. deals with inner structures of the main components, namely the core api and the development tools. as another major principle applied within generation theory is the use of ready made software wherever possible, the model's contents deal with artefacts framing the design of each such component. for the case of the reporting toolkit mainly xsl design and build-scripts will be modeled right here.

the ZF-row 5 usually captures some textual source code. expressed in terms of viewpoints, row 5 collects software artefacts based on some formal description.

the columns what, how and where are the most obvious ones, scoping artefacts in typical programming languages or more broader, computer languages. for the latter one terms like xml jump into focus.

As it is hard to imagine all single artefacts from this model's point of view the original idea of the framework for software architecture is extended a bit in mind, to find simple modeling conventions. the goal always has to be, classifying the main idioms, along with finding coding- or documenting-conventions to be expected, or identify them later on.

this very row is the one belonging to the programmer, that's where he is the expert.
an experienced programmer reuses it's knowledge at least. the more experienced he is the better defined his patterns of code structures will be.

the best point to start a project-model, leaned towards Zachman's ideas simply is give the main-items some names. within the model seen, they represent nothing but rows.

the names selected, rely on the one found in [Zachman 1996]. no changes to the namings as chosen there are made, although it's likely to happen.

the Zachman framework and it's rows. nothing but points of view.

of course, major modules, better say, standalone applications, defined as logical assemblys within one project, could be taken for naming as well.

but right now this idea is far away from being expressable.

within today's software development it is common sense, that features and requirements have to be collected and maintained in a systematic manner.

tools that are subject of investigation are bug tracking systems like mantis, issue trackers and uncountable systems of all kind. software like scarab enables the definition, implementation and extension of all types of issues, bugs are one example for. scarab e.g. comes along with a meta-model, developers can adjust for their own needs.

having that in mind, the use of UML could be extended for the configuration of such customizations, too. as a framework for smart model-driven generation would be useful anyway. it is obvious, that using UML like done for the ZF will move the focus to smart Profile-design. and the latter will have to be modeled again.

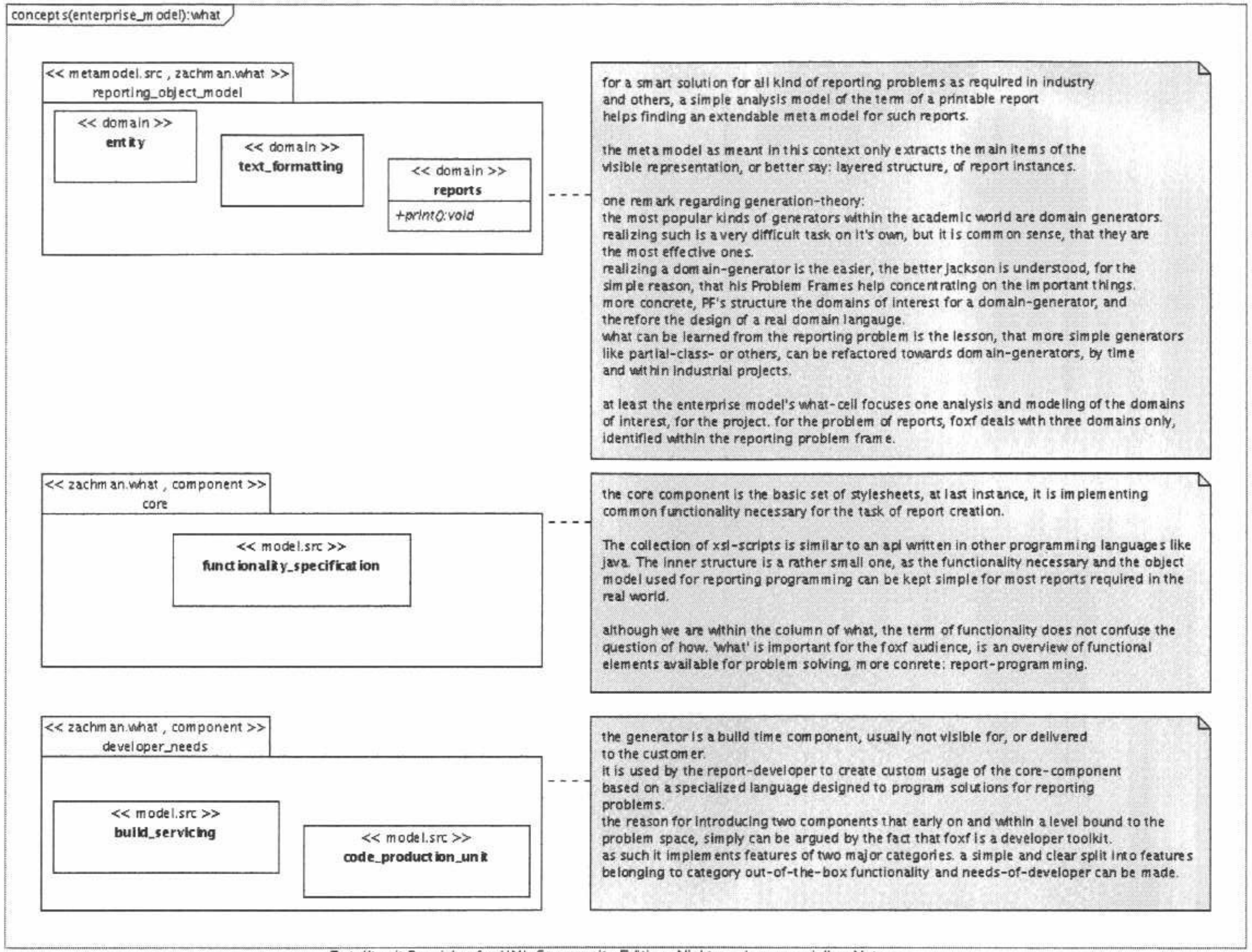
remarks on the different models:

for the distinction of the rows a short comparison is worth discussing. the system model is the first and most abstract classification with respect to the solution space. as such,service functions, or broader:
service-interfaces are things to be specified within the scope of ZF-row 3, something that the system developed provides as functionality to be used. the needs are located within the problem space, in terms of Zachman, rows 1 & 2. from the system designers point of view it is hard to predict the kind of patterns framing solutions realized by whatever programming language. the use of particular components-off-the-shelf (abbr. cots) frame the patterns and languages used, too. but anyway, within a model specifying nothing but the major artefacts classified as framing the software's architecture, typical functionality or all kinds of system-models scope the system's point of view.

more abstract again, the system model contains documents defining major components, service-implementations of service interfaces provided by cots and principle relationships between them.
the inner design of those components are bound to the tech. mod., that itself is influenced by decisions made earlier on, on questions of methods applied.

alternatively it can be remarked, that for the case of foxf row 3 defines low-level features to be used from developers, row 4 constrains their implementation. keeping that in mind will help understanding that patterns of row 3 may rely on simple design patterns, but beyond that, patterns as pattern-integrations of multiple patterns, and semantic enrichment scopes the artefacts contained in the system model.

originally Zachman did not talk
about patterns and how they
influence particular cells.
but anyway, patterns of all kinds
help ordering those cells
in a more general manner.



for a smart solution for all kind of reporting problems as required in industry and others, a simple analysis model of the term of a printable report helps finding an extendable meta model for such reports.

the meta model as meant in this context only extracts the main items of the visible representation, or better say: layered structure, of report instances.

one remark regarding generation-theory:
the most popular kinds of generators within the academic world are domain generators. realizing such is a very difficult task on it's own, but it is common sense, that they are the most effective ones.
realizing a domain-generator is the easier, the better Jackson is understood, for the simple reason, that his Problem Frames help concentrating on the important things. more concrete, PF's structure the domains of interest for a domain-generator, and therefore the design of a real domain langauge.
what can be learned from the reporting problem is the lesson, that more simple generators like partial-class- or others, can be refactored towards domain-generators, by time and within industrial projects.

at least the enterprise model's what-cell focuses one analysis and modeling of the domains of interest, for the project. for the problem of reports, foxf deals with three domains only, identified within the reporting problem frame.

the core component is the basic set of stylesheets, at last instance, it is implementing common functionality necessary for the task of report creation.

The collection of xsl-scripts is similar to an api written in other programming languages like java. The inner structure is a rather small one, as the functionality necessary and the object model used for reporting programming can be kept simple for most reports required in the real world.

although we are within the column of what, the term of functionality does not confuse the question of how. 'what' is important for the foxf audience, is an overview of functional elements available for problem solving, more conrete: report-programming.

the generator is a build time component, usually not visible for, or delivered to the customer.
it is used by the report-developer to create custom usage of the core-component based on a specialized language designed to program solutions for reporting problems.
the reason for introducing two components that early on and within a level bound to the problem space, simply can be argued by the fact that foxf is a developer toolkit.
as such it implements features of two major categories. a simple and clear split into features belonging to category out-of-the-box functionality and needs-of-developer can be made.

Erstellt mit Poseidon for UML Community Edition. Nicht zur kommerziellen Nutzung.

within the mtk, a toolkit embedding the foxf or other software's developer functionality, the concept of component means exchangeable pieces of software. exchangeable with respect to installation and the like.

getting big pieces of software, that usually are assembled by components themselves and carry hand-crafted code beyond that, makes it worth thinking about standardization of build-time processing.

tools like ant provide simple scripting means that can help solving all kind of tasks for packaging, versioning and others, of such components or whatever piece of software.

the application of ZF, using UML, targets the principle relationships between whatever components are developed, ready-made used etc.
right now this idea of using UML is in it's early stages. on the other hand a real-work proven library of ant scripts is available already.
for the further development of this profile along with the library itself, once established model driven generation approaches jump into eye as another application for generation to care about.

the enterprise model's classes of things usually are organized within what are known as domain-models. the difficulty of understanding lies within the term of domains.
a very useful definition of the term can be found in [Jackson '96]. more concrete, his concept of problem frames helps finding s systematic approach for organizing the artefacts of interest from the analyzers point of view.
out-of-the box functionality means that part of the system, or say the software's architecture, the customer looks on. beyond that the developer has to do daily work to implement reports, or any other kind of software components. introducing components already within the problem space, simplifies clean generation of development-process implementing code.

code for build-scripting
and the like is code on
it's own. so it's obvious,
that it's subject of
generation, too.

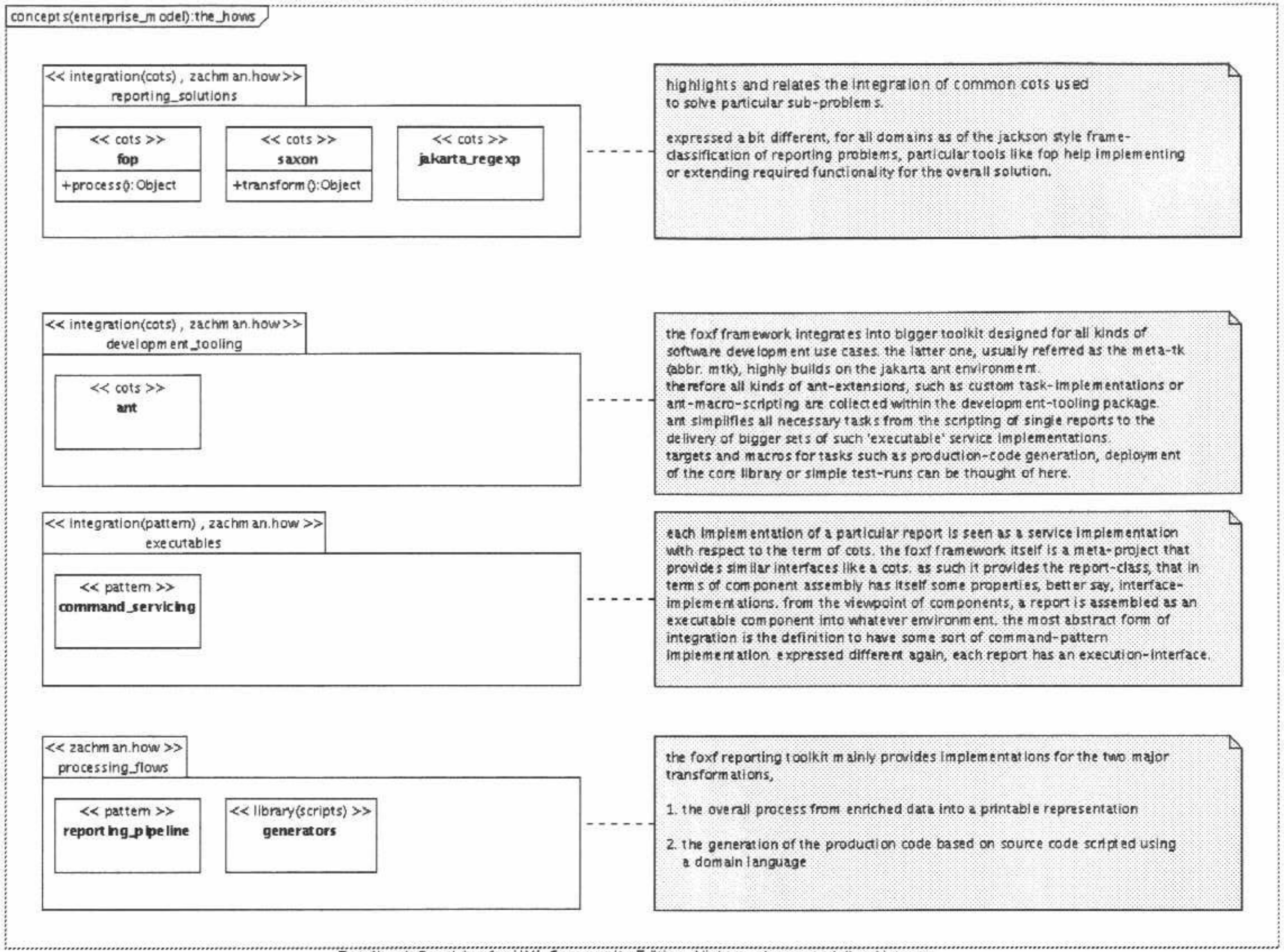
the most effective approach of generation is seen in the use of domain languages. for the scope of enterprise models, the design of domain models, resulting in languages like that, is a hurdle to jump.

it is hard to guess, what it will take, to get a general method for domain-modeling, as it is a big research topic itself.

Jackson does nothing but highlighting their importance and how they can be related.

but back to stuff like build-scripts, it is pretty clear, there are domains modeling applications of software engineering additionally, and like for any other type of problem. so better handle them like software-problem on their own.

why taking a different approach for the generation of build-time-, than for production-code. get the idea?!



highlights and relates the integration of common cots used to solve particular sub-problems.

expressed a bit different, for all domains as of the jackson style frame-classification of reporting problems, particular tools like fop help implementing or extending required functionality for the overall solution.

the foxf framework integrates into bigger toolkit designed for all kinds of software development use cases. the latter one, usually referred as the meta-tk (abbr. mtk), highly builds on the jakarta ant environment.
therefore all kinds of ant-extensions, such as custom task-implementations or ant-macro-scripting are collected within the development-tooling package.
ant simplifies all necessary tasks from the scripting of single reports to the delivery of bigger sets of such 'executable' service implementations.
targets and macros for tasks such as production-code generation, deployment of the core library or simple test-runs can be thought of here.

each implementation of a particular report is seen as a service implementation with respect to the term of cots. the foxf framework itself is a meta-project that provides similar interfaces like a cots. as such it provides the report-class, that in terms of component assembly has itself some properties, better say, interface-implementations. from the viewpoint of components, a report is assembled as an executable component into whatever environment. the most abstract form of integration is the definition to have some sort of command-pattern implementation. expressed different again, each report has an execution-interface.

the foxf reporting toolkit mainly provides implementations for the two major transformations,

1. the overall process from enriched data into a printable representation

2. the generation of the production code based on source code scripted using a domain language

the term of cots is defined in different ways. some works are talking about commercial off-the-shelf, but the author's interpretation is taken from [Aßmann 2003], who calls them components-off-the-shelf.

besides that there are works caring about the differences between commercial and open-source. it is just not that important for the ZF for the time being, as discussing cots in all their variant forms is subject work a document on it's own.

the mtk takes the usage of cots as an inner success guaranteeing method for code generation. it can be argued at least, that particular cots play one or the other role for the build-time artefacts generated from a ZF-UML-profile application.

the reporting problem's enterprise model, aka. the foxf implementation guidelines, mainly relate items useful for the solution. useful in terms of implementing the overlying system's features.

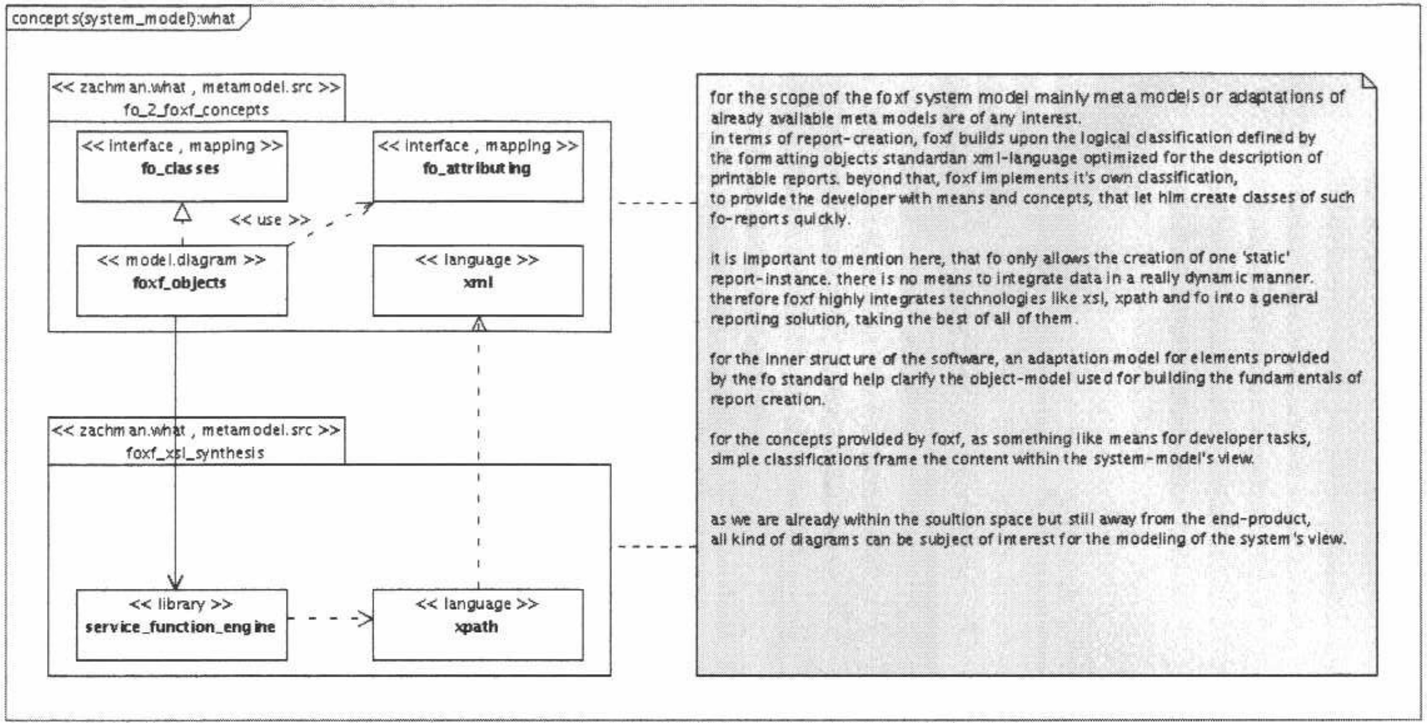
it is not important to care about the features themselves, but relating ready-made components and whenever possible, to highlight how they implement features as required already. at least to some extent features identified as useful one's for a general solution to the reporting problem are interest of modeling.

the central application of cots-usage gets it's motivation from a completely different fact. just imagine the patterns they implement.

the enterprise model's column of how is the one closest to what Jackson calls a PT's method, the approaches useful for the solution's implementation. as those might be as different like the problems they solve, simple things like cots used, patterns applied or defined, are the most obvious items to be contained within the project's enterprise how aspects.

these two items, cots & pattern, modeled as used for a custom software package, highly simplify tasks like deployment, or more general, build-time tasks of all kinds. be it to script them manually, or generate them.

btw. for the distinction of build-time and runtime look into [Bassett 1996].



for the scope of the foxf system model mainly meta models or adaptations of already available meta models are of any interest.
in terms of report-creation, foxf builds upon the logical classification defined by the formatting objects standardan xml-language optimized for the description of printable reports. beyond that, foxf implements it's own classification, to provide the developer with means and concepts, that let him create classes of such fo-reports quickly.

it is important to mention here, that fo only allows the creation of one 'static' report-instance. there is no means to integrate data in a really dynamic manner. therefore foxf highly integrates technologies like xsl, xpath and fo into a general reporting solution, taking the best of all of them.

for the inner structure of the software, an adaptation model for elements provided by the fo standard help clarify the object-model used for building the fundamentals of report creation.

for the concepts provided by foxf, as something like means for developer tasks, simple classifications frame the content within the system-model's view.

as we are already within the soultion space but still away from the end-product, all kind of diagrams can be subject of interest for the modeling of the system's view.

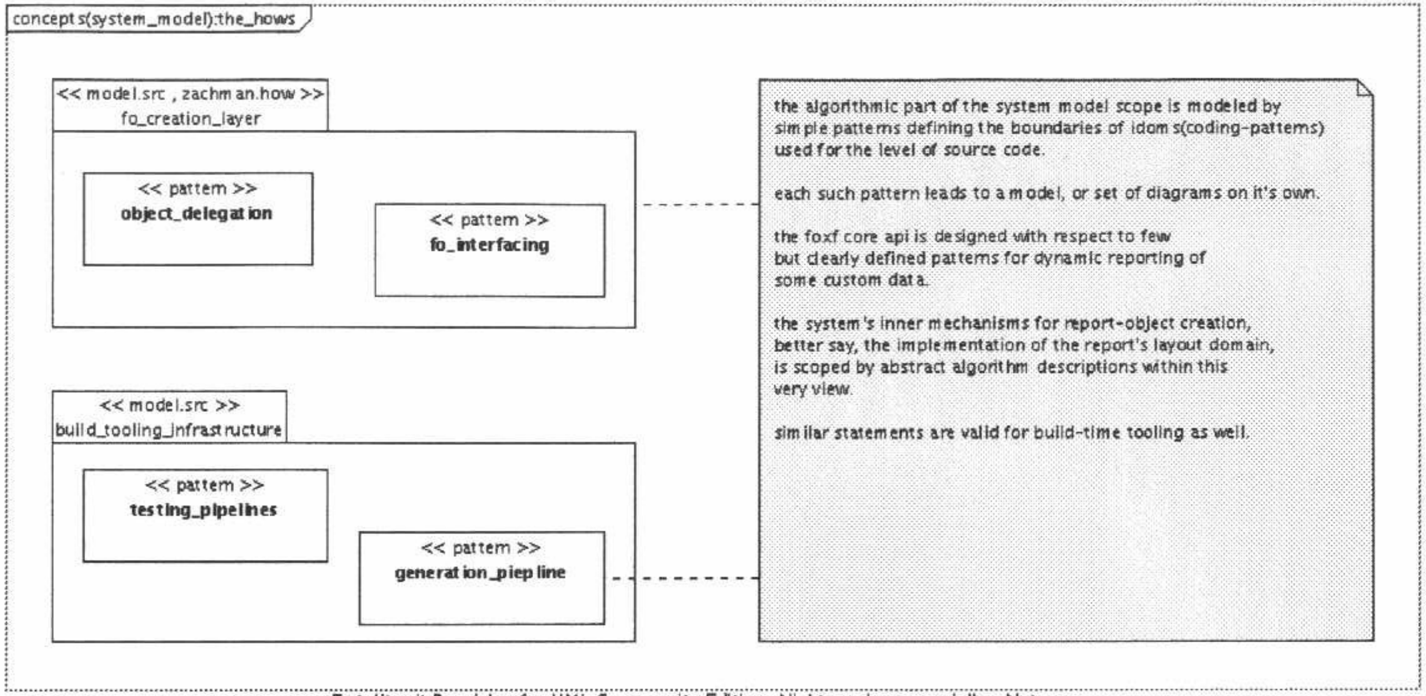
Erstellt mit Poseidon for UML Community Edition. Nicht zur kommerziellen Nutzung.

from the perspective of generator programming. with sources taken from the designer's view, model-driven generation like found in [Kleppe et.al. 2003] jumps into focus again.
it could be said, that from here on, platform specific models (abbr. psm) play a major role. the difficulty just is, the still missing understanding for the term of model-generators, meaning: a useful approach for this kind of generation is still lack of existence. at least as far as the author's state of knowledge is concerned.

the work of [Buschmann et. al. 1996] introduces the term of pattern-systems. system here, best is understood as a systematic approach of applying multiple patterns to form a bigger whole.

with respect to the ZF system models, that's exactly what is required. relate applications of patterns, defined in detail on deeper views, into the bigger whole again.

for the ideal of taking pattern-formalization as a major fundamental for generation-theory, systems like that one, will be needed, to help solving brain-teasers like the integration & assembly of patterns into new one's.



the algorithmic part of the system model scope is modeled by simple patterns defining the boundaries of idoms(coding-patterns) used for the level of source code.

each such pattern leads to a model, or set of diagrams on it's own.

the foxf core api is designed with respect to few but clearly defined patterns for dynamic reporting of some custom data.

the system's inner mechanisms for report-object creation, better say, the implementation of the report's layout domain, is scoped by abstract algorithm descriptions within this very view.

similar statements are valid for build-time tooling as well.

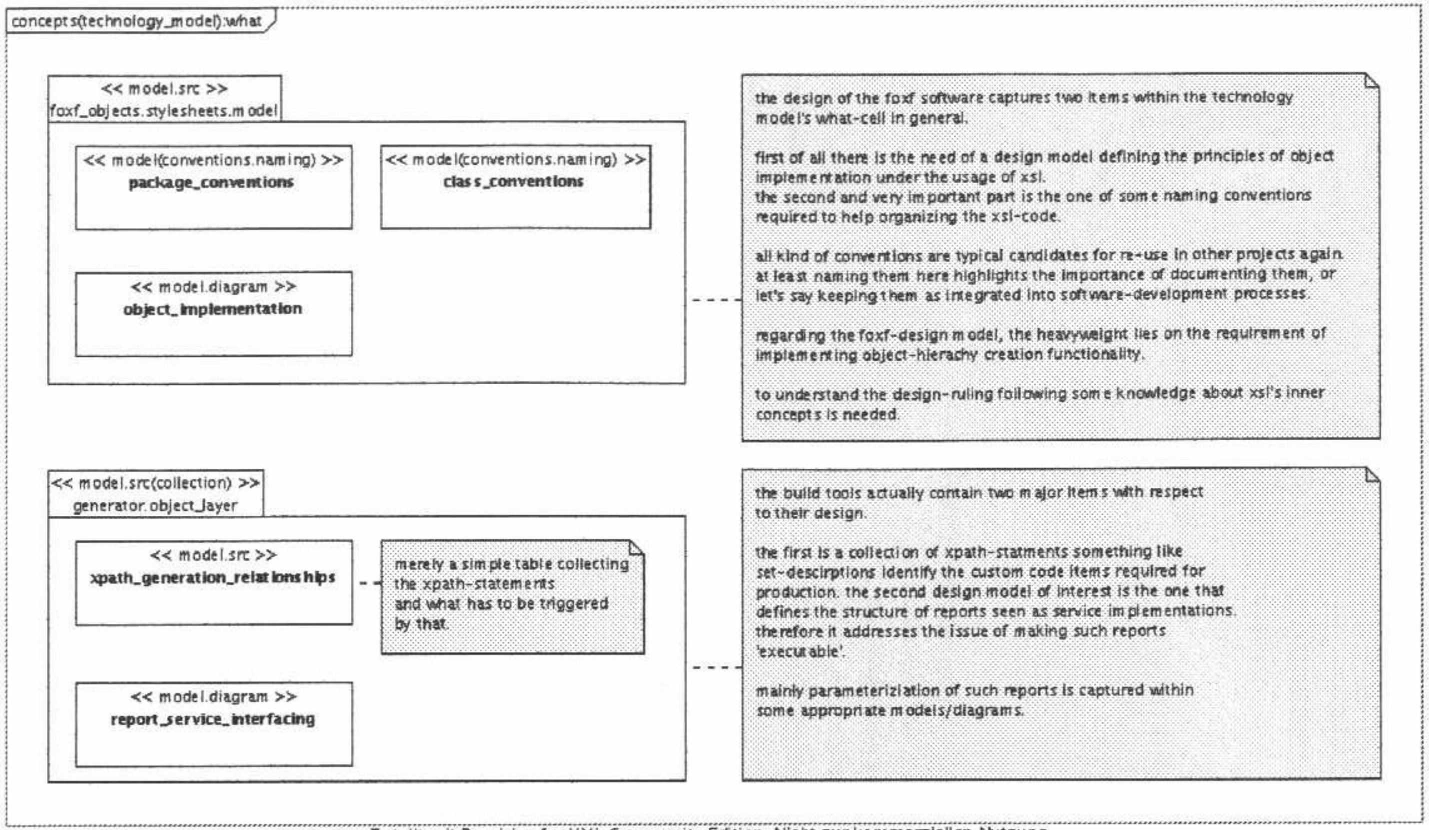
the distinction between all the different scopes defined within the ZF's cells is an issue that could raise never ending discussions. to avoid confusion, system designs are understood as that part of software development that bridge the gap from problem to solution space.

within here first decisions, influencing the software's inner design are located. remember the comments made on the system-model package earlier.

the application of pattern systems simplify the systematic necessary to use models out of this row in general, the how-cell in particular. for a meta-level model like this exemplification of a ZF-model, classifications, building up such systems, are a means to help organizing the system level across projects. for projects themselves the all important hint for real-life is, try building systems by relating single fine-grained patterns by documents and models under the view of row 3. but don't confuse 'meta-level models' by meta-models.

that very model, regardless of the column contains all kinds of artefacts capturing the boundaries of the framework design.
as the resulting code still is one stage ahead, graphical representations most probably are those being defined on this view.

remark:
to get a detailed overview of languages useful for the different cells have a detailed look into [Hay 2002]


concepts(technology_model):what
<< model.src >>
foxf_objects.stylesheets.model
<< model(conventions.naming) >>
package_conventions
<< model(conventions.naming) >>
class_conventions
<< model.diagram >>
object_implementation
<< model.src(collection) >>
generator.object_layer
<< model.src >>
xpath_generation_relationships
<< model.diagram >>
report_service_interfacing

merely a simple table collecting
the xpath-statements
and what has to be triggered
by that.

the design of the foxf software captures two items within the technology model's what-cell in general.

first of all there is the need of a design model defining the principles of object implementation under the usage of xsl.
the second and very important part is the one of some naming conventions required to help organizing the xsl-code.

all kind of conventions are typical candidates for re-use in other projects again. at least naming them here highlights the importance of documenting them, or let's say keeping them as integrated into software-development processes.

regarding the foxf-design model, the heavyweight lies on the requirement of implementing object-hierachy creation functionality.

to understand the design-ruling following some knowledge about xsl's inner concepts is needed.

the build tools actually contain two major items with respect to their design.

the first is a collection of xpath-statments something like set-descirptions identify the custom code items required for production. the second design model of interest is the one that defines the structure of reports seen as service implementations. therefore it addresses the issue of making such reports 'executable'.

mainly parameterizlation of such reports is captured within some appropriate models/diagrams.

Erstellt mit Poseidon for UML Community Edition. Nicht zur kommerziellen Nutzung.

the techn.-model deals with the inner design of software within the boundaries defined by the sys.-model. for the column of what the UML-Profile for ZF captures design-models and -conventions.

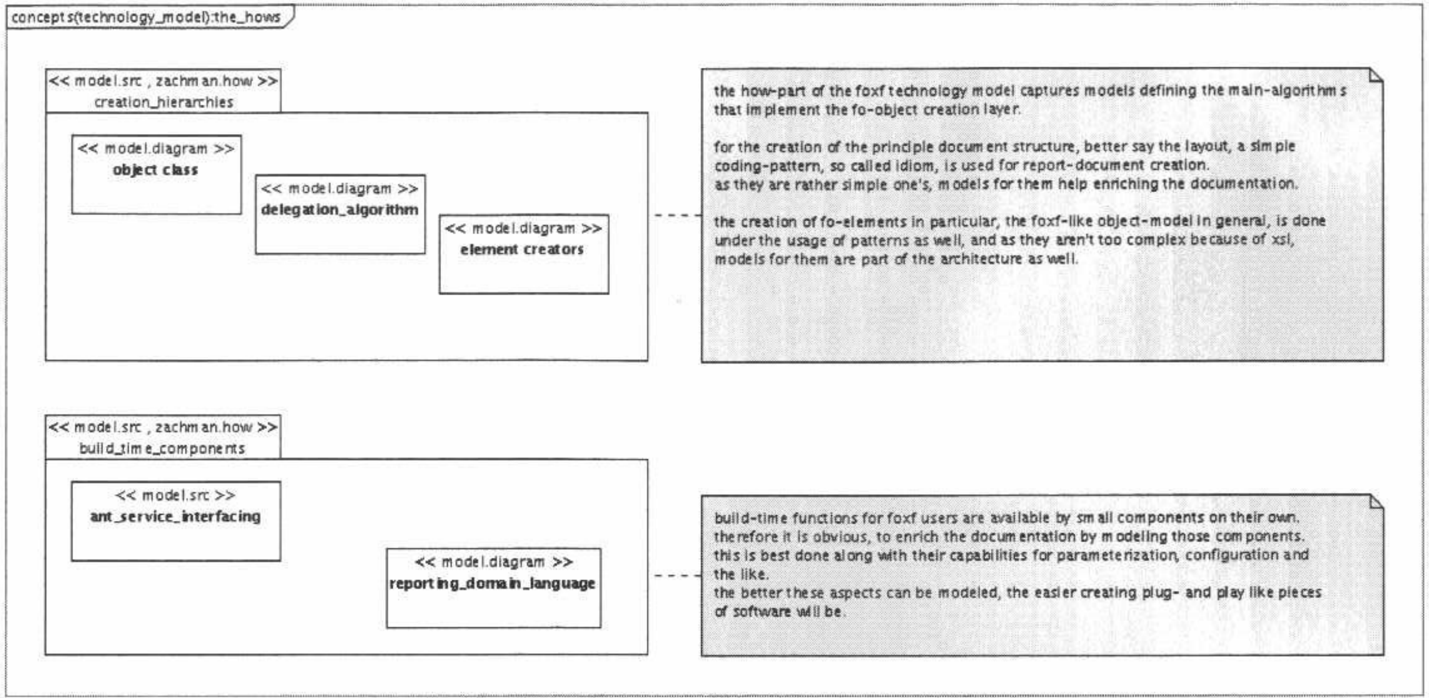
diagrams declared on this layer are the one's with the highest importance for 'real' code generation, besides the designs themselves, modeling-, naming-, documentation- or whatever kind of conventions have an aspect-like influence to the code itself.

tools for coding conventions e.g. exist already, and usually are easy to integrate into build tools like ant, or IDE's like eclipse. depending on their flexibility former formalization of such ruling is a subject of discussion on it's own again.

one remark on stereotypes:
why not extending them by principles
close to predicate logic.
still an idea in mind.

the use of stereotypes. better systems of stereotypes, a.ka. UML-Profiles, raises another problem. as the particular profiles used for application modeling depend on the configuration established in a ZF-model , such model-files have to be pre-generated and/or updated. Solving that very problem moves the focus for a while. first of all, it is worth mentioning, that aproblem like this is a very common one within the topic of generation anyway.

At this point in time a short sideview to [Herrington 2003] shall help getting a bit of light into darkness. Herrington himself knows six kinds of generators, that are something like well-known patterns for code-generators. He talks about code-mungers. inline-code-expanders, mixed-code-generators, that all take 'real' source-code for input. furthermore he knows about partial-class, full-tier and domain-generators. to get an idea of their different properties check out his work. Although his work is a very useful one, he makes one big mistake. for him, model-generators are an example of full-tier-gen. The work-out of the differences would be enough for another document like this. But to provide a hint, think about the role of UML-profiles. Besides all that the history of foXf has shown how to get to dom.-gen. by time.



the how-part of the foxf technology model captures models defining the main-algorithms that implement the fo-object creation layer.

for the creation of the principle document structure, better say the layout, a simple coding-pattern, so called idiom, is used for report-document creation.
as they are rather simple one's, models for them help enriching the documentation.

the creation of fo-elements in particular, the foxf-like object-model in general, is done under the usage of patterns as well, and as they aren't too complex because of xsl, models for them are part of the architecture as well.

build-time functions for foxf users are available by small components on their own.
therefore it is obvious, to enrich the documentation by modeling those components.
this is best done along with their capabilities for parameterization, configuration and the like.
the better these aspects can be modeled, the easier creating plug- and play like pieces of software will be.

Erstellt mit Poseidon for UML Community Edition. Nicht zur kommerziellen Nutzung.

the big question:
how to generalize
algorithms, better say
functionality.
still another brainteaser,
but defining the main
patterns applied, should
be a starting point.

the inner design of functional aspects of a software in general may be the most critical aspect for the finite goal of smart code generation. graphical languages could be useful to create particular types of algorithms. UML-like behavioral diagram types are one example for those kind of languages, again with respect to the mentioned work by David Hay. althought their use is very restricted.
the most promising approach to close some gaps seems to be the Action Semantics Language part of the Executable-UML [Mellor Balcer 2002]. just discussing that particular stuff in detail raises side-views to principles like [GenVoca].
still a brainteaser.

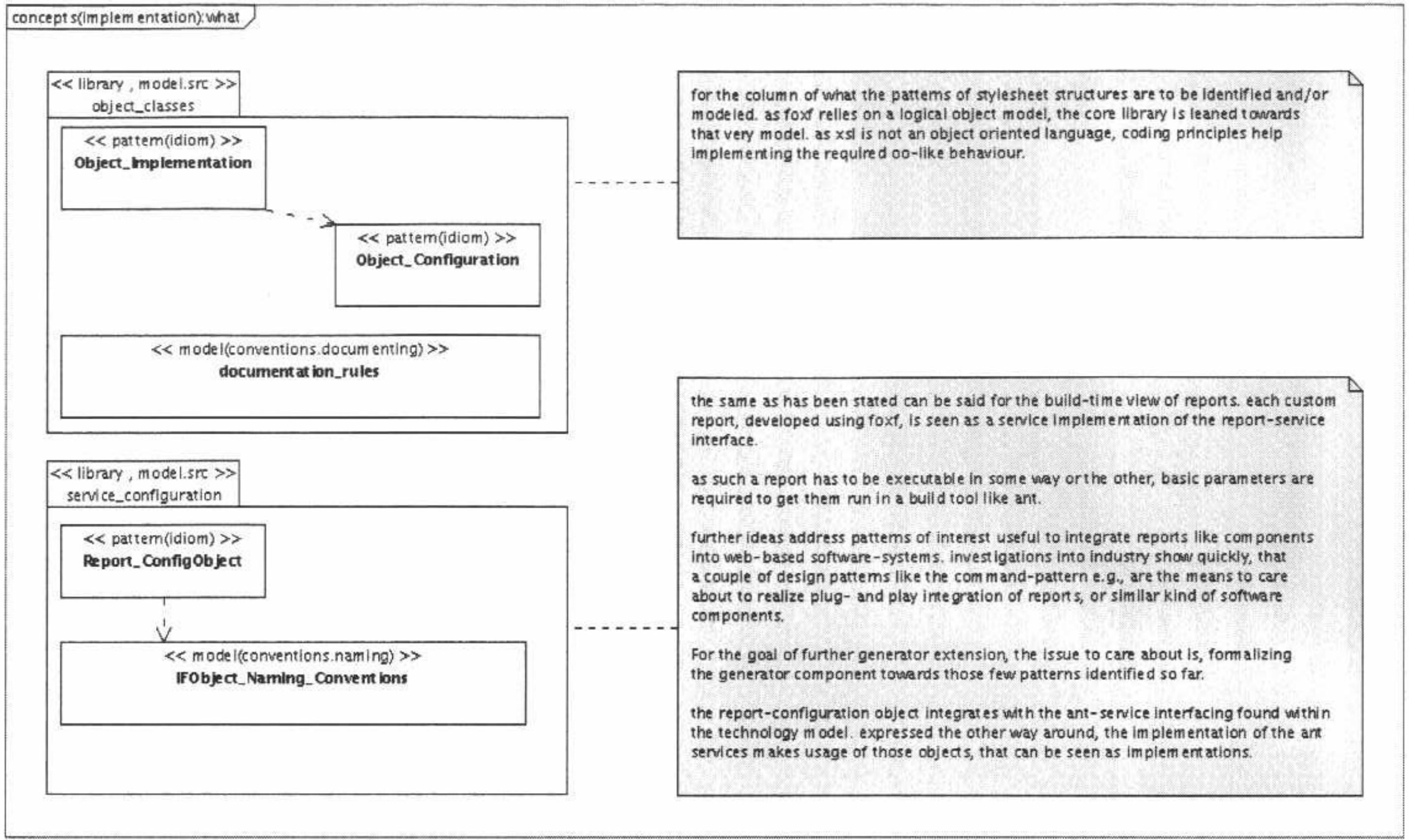
some thoughts on Executable UML:
the Action-Semantics are really worth studying, but what is an executable model good for?! the customer wants a running software and nothing else.

this very cell highlights the difficulty of overall code-generation. the requirements with respect to algorithms. the only succesful way here may be the formalization of design-patterns and their implementations within components used.

diagrams like all behavioral types only can be a first stage for functional aspects of a design or implementation. at least it is more than obvious, that formal approaches for structural aspects help minimizing required hand-craft for the functional one's.

of course Jackson is worth mentioning again. graphical languages can be useful within the scope of particular problems, but for sure not every problem around.

to get another idea why pure graphical, let's call it visual-programming, will never come true, look into [Schiffer 1997]



for the column of what the patterns of stylesheet structures are to be identified and/or modeled. as foxf relies on a logical object model, the core library is leaned towards that very model. as xsl is not an object oriented language, coding principles help implementing the required oo-like behaviour.

the same as has been stated can be said for the build-time view of reports. each custom report, developed using foxf, is seen as a service implementation of the report-service interface.

as such a report has to be executable in some way or the other, basic parameters are required to get them run in a build tool like ant.

further ideas address patterns of interest useful to integrate reports like components into web-based software-systems. investigations into industry show quickly, that a couple of design patterns like the command-pattern e.g., are the means to care about to realize plug- and play integration of reports, or similar kind of software components.

For the goal of further generator extension, the issue to care about is, formalizing the generator component towards those few patterns identified so far.

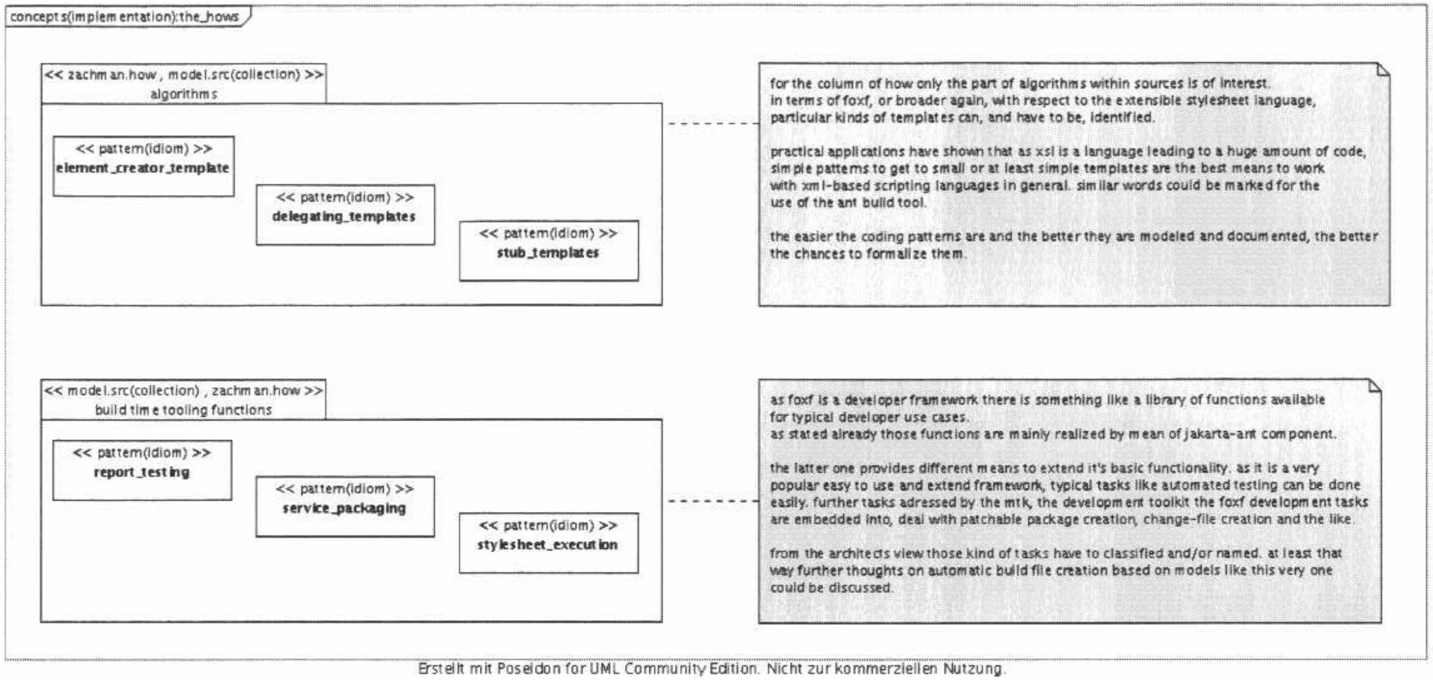
the report-configuration object integrates with the ant-service interfacing found within the technology model. expressed the other way around, the implementation of the ant services makes usage of those objects, that can be seen as implementations.

coding- and documenting conventions are simple means, that help organizing the chaos. from the software architect's point of view, means, the person organizing the structure of artefacts to be collected within projects, needs simple means defined, the programmer uses when doing his work.

beyond all kind of conventions so called idioms, patterns for source-code with respect to concrete programming languages(e.g. C, xsl, ...), are a popular and effective means to further formalize the process of software production.

to get a better idea about idioms have a look into [Buschmann et. al. 1996] for the architect as well as the programmer identifying those idioms and relating them into a simple model, will simplify knowledge-reuse in future projects.

idioms are those kind of patterns found on the lowest level in terms of ZF-rows, as they deal with the structure of code.
their formalization could turn out to be the easiest one, as the underlying language used, already relies on some kind of formalism. from there on, as idioms scope functionality as well, further thoughts on the integration of Action Semantics could be worth writing down on their own.



for the column of how only the part of algorithms within sources is of interest.
in terms of foxf, or broader again, with respect to the extensible stylesheet language, particular kinds of templates can, and have to be, identified.

practical applications have shown that as xsl is a language leading to a huge amount of code, simple patterns to get to small or at least simple templates are the best means to work with xml-based scripting languages in general. similar words could be marked for the use of the ant build tool.

the easier the coding patterns are and the better they are modeled and documented, the better the chances to formalize them.

as foxf is a developer framework there is something like a library of functions available for typical developer use cases.
as stated already those functions are mainly realized by mean of jakarta-ant component.

the latter one provides different means to extend it's basic functionality. as it is a very popular easy to use and extend framework, typical tasks like automated testing can be done easily. further tasks adressed by the mtk, the development toolkit the foxf development tasks are embedded into, deal with patchable package creation, change-file creation and the like.

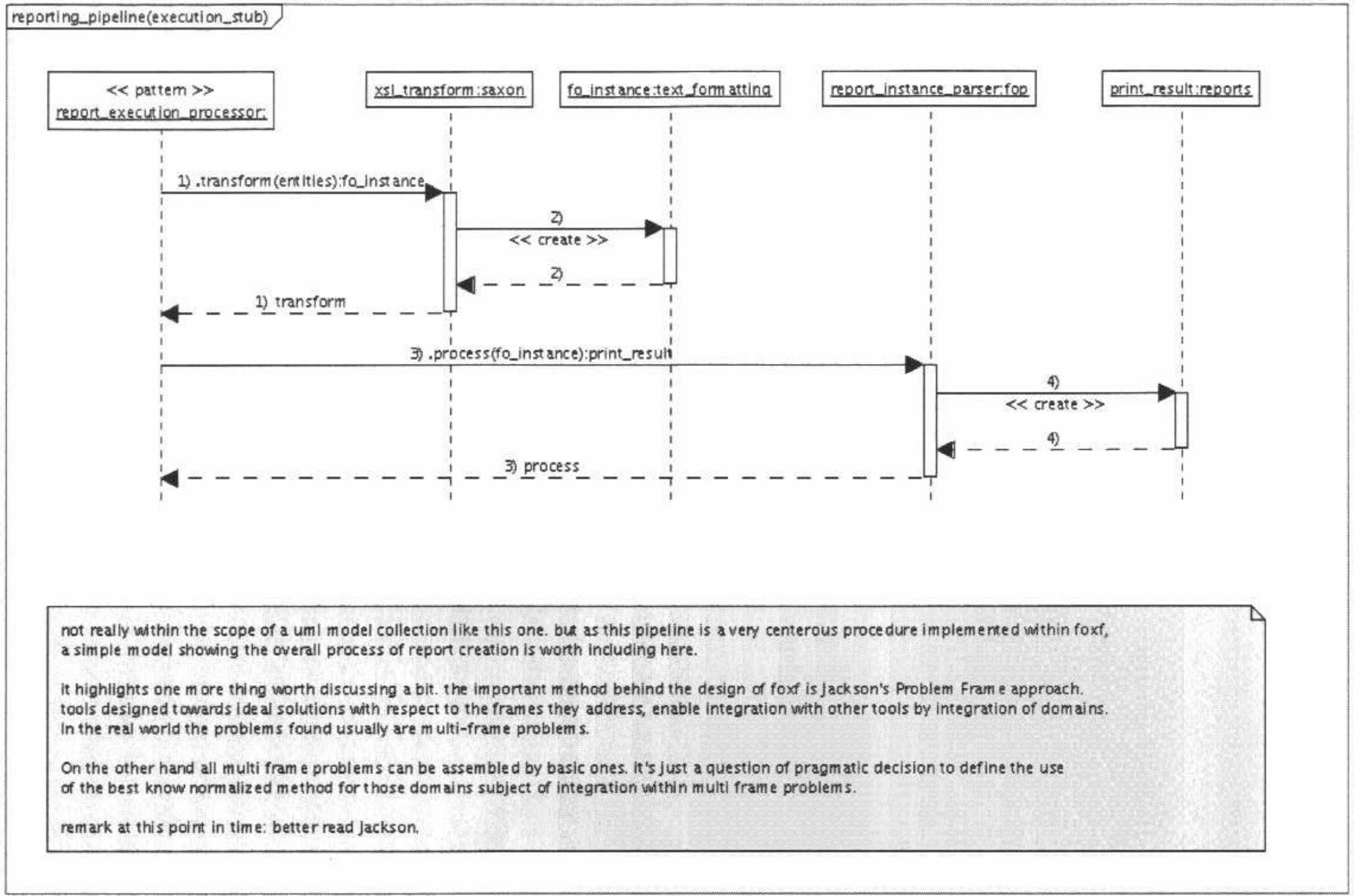
from the architects view those kind of tasks have to classified and/or named. at least that way further thoughts on automatic build file creation based on models like this very one could be discussed.

Erstellt mit Poseidon for UML Community Edition. Nicht zur kommerziellen Nutzung.

for the implementation of functional aspects, similar languages as for the what items are subject of usage. although usually only 'real' programming languages should be expected here.

the what cell, implementing the things of interest, classes e.g. data-describing languages are within the scope additionally. in general it can be stated, that both columns can be found within one piece of code at the same time. ZF therefore further helps distincting between the generation of structural and functional aspects of source code.

most probably. formalization of idiom-usage will be one of the major secrets behind automated software production. the better writing them down, the better and easier the design of an overall generation-toolkit, or let's say framework, for another time.


reporting_pipeline(execution_stub)
<< pattern >>
report_execution_processor:
xsl_transform:saxon
fo_instance:text_formatting
report_instance_parser:fop
print_result:reports
1) .transform(entities):fo_instance
2)
<< create >>
2)
1) transform
3) .process(fo_instance):print_result
4)
<< create >>
4)
3) process

not really within the scope of a uml model collection like this one. but as this pipeline is a very centerous procedure implemented within foxf, a simple model showing the overall process of report creation is worth including here.

it highlights one more thing worth discussing a bit. the important method behind the design of foxf is Jackson's Problem Frame approach. tools designed towards ideal solutions with respect to the frames they address, enable integration with other tools by integration of domains. In the real world the problems found usually are multi-frame problems.

On the other hand all multi frame problems can be assembled by basic ones. it's just a question of pragmatic decision to define the use of the best know normalized method for those domains subject of integration within multi frame problems.

remark at this point in time: better read Jackson.

one more secret to plug-and-play like software construction: smart integration of problem-domains that serve as integrators within multi-frame problems.

the inclusion of this model does nothing but demonstrating the importance of Jackson another time. reporting problems e.g. embed into two more PF's as found within software systems. in the real world, the process of report-creation is preceded by some kind of data-queries or data-mining and the like. single transformations like those two are integrated into some pipeline, like the one above. the latter one, a problem frame on it's own again.

Summary

as shown, generation-theory is a rather complex subject and hard to get. most probably it's not even established, as available approaches somehow related are uncountable and usually pretty different. to say it different, putting them into a work like that one, is impossible by now. Under the guideline of the Zachman-Framework for SW-Architecture a mere rough overview could be given. Using his ideas means transforming between cells. for sure that is the major reason, to take it as a base for the collection of some ideas. because of it's young age, ideas on generative programming are far away from being complete. works like the Jackson one's help developers anywhere. for generators there is one question be ~~th~~ kept in mind. if, and there are quite some reasons to imagine the fact, the amount of software problems is an endful one, overall code-generation, may be possible.

On the other hand the Zachman Framework has 30 cells, so there is quite some work ahead.

References

[Aßmann 2003]
Invasive Software Composition
Uwe Aßmann; Springer, Berlin

[Bassett 1996]
Framing Software Reuse: Lessons from the Real World
Paul Bassett; Prentice Hall

[Buschmann et. al. 1996]
A System of Patterns. Pattern-Oriented Software Architecture
Frank Buschmann and others; John Wiley and Sons Ltd

[Kerrington 2003]
Code Generation in Action
Jack Kerrington; Manning

[Robertson 2000]
Requirements Patterns via Events-Use Cases
Suzanne Robertson

[GenVoca]
Batory Chen Robertson Wang 2000; Design Wizards and Visual Programming Environments for GenVoca Generators
Batory Geraci 1997; Composition Validation and Subjectivity in GenVoca Generators
.. and others ..

[Hay 2002]
Requirements Analysis
David Hay; Prentice Hall

[Kleppe et. al. 2003]
MDA Explained. The Model Driven Architecture: Practice and Promise
Anneke Kleppe, Jos Warmer, Wim Bast

[Mellor Balcer 2002]
Executable UML. A Foundation for Model Driven Architecture
Stephen J. Mellor, Marc J. Balcer; Addison-Wesley Professional

[Schiffer 1997]
Visuelle Programmierung. Grundlagen und Einsatzmöglichkeiten
Stefan Schiffer; Addison Wesley Verlag

[Zachman 1996]
Enterprise Architecture: The issue of the Century
John A. Zachman; Zachman Institute for Framework Advancement

[Jackson 1996]
Requirements and Specifications. A Lexicon of Software Practice, Principles and Prejudices
Michael A. Jackson; Addison-Wesley Professional

[Jackson 2000]
Problem Frames. Analyzing and Structuring Software Development Problems
Michael A. Jackson; Addison-Wesley Professional

better read Jackson.
if done so,
read it again.
until you think you
understood him.
start all over then.