The present work was submitted to the Research Group Software Construction

of the Faculty of Mathematics, Computer Science, and Natural Sciences

Master Thesis

An Incremental Code Generator for Heterogeneous Software and Infrastrucutre

Ein inkrementeller Code Generator für heterogene Software und Infrastruktur

presented by

Ralph Geerkens

Aachen, May 9, 2017

Examiner

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Supervisor

Dipl.-Inform. Andreas Steffens, M.Sc.

Statutory Declaration in Lieu of an Oath

The present translation is for your convenience only.
Only the German version is legally binding.

I hereby declare in lieu of an oath that I have completed the present Master's thesis entitled

An Incremental Code Generator for Heterogeneous Software and Infrastrucutre

independently and without illegitimate assistance from third parties. I have use no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Official Notification

Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whosoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 to 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

I have read and understood the above official notification.

Eidesstattliche Versicherung

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Masterarbeit mit dem Titel

An Incremental Code Generator for Heterogeneous Software and Infrastrucutre

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, May 9, 2017 (Ralph Geerkens)

Belehrung

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicher ung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen.

Aachen, May 9, 2017 (Ralph Geerkens)

Abstract

Using modular software architectures, which favours separation of concern, results in more complex software development environments. With DevOps not only software developers but also operationals will work in the same environment making it more heterogeneous. A code generator can reduce the overhead of creating a new development environment, which corresponds to the architecture of the software system and which has to address the concerns of multiple stakeholders. Incremental usage and generation of an exemplary application make the generator applicable at any time during the software development process and also by inexperienced developers and operationals. This thesis presents the architecture and concepts of a code generator, which addresses those issues, and has shown by an evaluation, that it fastens the creation of development environments and its very flexible in its usage.

Contents

List of Tables

List of Figures

List of Source Codes

1. Introduction

Real programmers don't comment their code. It was hard to write, it should be hard to understand.

ANONYMOUS

Contents

Reuse in software engineering is important for companies to reduces software production and maintenance cost, to deliver software faster and to increase the quality of software [Som10]. Code modules, documentation, test data, requirements, design, code, architectures and whole systems or subsystems as reusable parts for software [Rin97]. Component-based and service-oriented software engineering are two ways to reuse software [Vli08]. Domain-driven design proposes the separation of the domain related parts of a software system from the remaining system [Eva03]. There are multiple domain-centric architectures, which help to create software systems following this ideas [Dca]. Microservices are an software architecture which breaks a software system into small and autonomous services [New15]. Domain-driven design and microservices are about separation of concern and shall help to reduce the complexity of software systems. Architecture patterns are a way to reuse the same architecture for multiple systems. The domain-centric architecture patterns and service-oriented architectures patterns in general can be reused to achieve separation of concerns. Applying both ideas or similar ones results in more complex development environments containing many projects. Setting up a new development environments for a service, which contains separate projects for the domain and technology, will become a constantly repeating task.

DevOps integrates development and operations [Wil+16]. Continuous delivery and deployment are two goals, which shall be achieved by it. Using the concept of infrastructure as code is important to achieve those goals but it adds additional complexity to the development environment of a software system. The whole environment becomes more heterogeneous and it is even more costly to create one for small services.

At this point code generators can be helpful to create a development environment for a small service and its infrastructure, which is developed by following the principles of domain-driven design . Good support of the used architecture pattern and their implementation within a company can avoid manual adaptations of the generated code. But the complexity of the development environment still remains and makes it hard for software developers to work with it, who are inexperienced with the architecture and technologies in such an environment. An exemplary application, which uses the same one, can illustrate the architecture patterns and their components. The same code generator can be used to make this such an application available to each developer. In software system with a domain-centric architecture, the technology can change for different software system. The software developer needs to be able to use the code generator incrementally and to compose a custom development environment based on requested technologies.

So in this thesis an incremental code generator is presented for the generation of a heterogeneous development environments in compliance to an architecture pattern, which contains software and infrastructure projects.

1.1. Structure of this Thesis

In the next chapter the motivation for a code generator and the detailed problem are presented. The whole thesis was accompanied by a case study at KISTERS AG. A set of requirements derived from interviews with employees at KISTERS AG will be also presented. Afterwards in chapter 3 existing code generators are compared and matched to the requirements. Chapter 4 introduces architecture viewpoints described by Rozanski and Woods and extends it by a model for the software development environment. Based on a better understanding of the development environment in chapter 5 an code generator architecture and concepts are presented. An implementation of those architecture and concepts for the KISTETRS AG is presented in chapter 6. This implementation is then evaluated according to the requirements and by users and developers of the code generator in chapter 7. The chapter is concluded by a broader discussion about the code generator. Finally in 8 the results are summarized and future work for improvement of the code generator is presented.

2. The Motivation and Requirements for a Code Generator

Don't panic!

DOUGLAS ADAMS

Contents

In the introduction three bigger requirements regarding a code generator were already mentioned. Creation of an development environment supporting the architecture of the software system, incremental code generation and being able to generate an exemplary application. Everyone of those aspects will be explained in more detail in this chapter but first the scope of such a code generator shall be explained. Finally detailed requirements of a code generator are presented.

Every company or organisation has its own needs. IT Governance shall help to make IT related decisions in alignment with the business needs [Dub+08]. For software development domain-specific governance exists. So how software is developed in each company may differ. For this thesis the scope of the code generator is its application in a single company. Problems or requirements resulting from software development processes from different companies does not have to be considered. Software developers and operationals are working together in a single company, but maybe in different teams.

2.0.1. Corporate-wide Architecture Patterns

Patterns provide a solution to specific problems in software development. Architecture patterns are a way to reuse software architecture for system having the same or similar problems. Software architectures following patterns may be easier to talk about with colleagues than those which do not.

When an architecture pattern is implemented decisions have to be made. First technologies for the implementation have to be picked. Then the components have to be organized physically in the development environment. A naming strategy should be applied for the components, the technology and the physical organization, making it for developer easier to find their way in the development environment. When software development teams in a company face similar problems regarding the architecture using them same architecture pattern can have a synergistic effect. If a common naming strategy, code organization and technology is used, developers can better make use of knowledge gained from former software projects. Additionally best practices regarding the architecture and the implementation of the components can be shared more easily. Corporate governance for software development, e.g. regarding the implementation of security related functions, can be more specific because of a common architecture and technology. If it is more specific and the impediment to implement it is smaller, the application of governance rules increases.

In this thesis the Ports and Adapters architecture pattern (PAP) is used as representative for all architecture patterns. A naming strategy and codeline organization for the PAP will be implement and utilized for code generation.

The Ports and Adapters pattern is a domain-centric architecture pattern [Coc]. It was first introduced by Alistair Cockburn under the name Hexagonal Architecture in 1995. Domain driven design is a design principle focusing on modelling the problem domain and using the same model within the implementation of the application [Eva03] A good separation of technology and domain makes the implementation easier and the model more independent from technology. Three different architecture pattern putting the domain at the centre separating it from technology are the Ports and Adapters, the Onion and Clean Architecture Pattern [Dca]. Compared to MVC or the three-tier architecture pattern domain-centric architecture patterns have more components. Because the domain has to be coherent the software systems are smaller. Independent of the kind of domain-centric architecture the basic idea is always the same, keeping the domain at the centre and keeping it separated from technology, which is needed to provide or consume services.

In figure 2.1 a digram of the PAP and its components is shown. Like other domain-centric architecture pattern the domain includes the entities and the use cases are at the core of the pattern. Interface separate the domain from the layer from the technical parts of the software system. These interfaces are called ports. Implementations of the interfaces are realizations of the adapter pattern [Gam+95] and are called adapters. Ports and adapters can distinguished as either primary and secondary. Primary adapters and ports provide services to other parts of the system or external systems by delegating to the domain layer. A REST interface is a often used as primary adapter. Secondary adapter ports consume other services and provide the result to the domain layer. A database adapter is a often used secondary adapter. By using dependency injection the domain layer has no dependencies on the surrounding adapters. Additionally the architecture has a service layer responsible for security and transaction management. By using the Ports and Adapters architecture pattern small and cohesive functional software unit can be created. The PAP can be used to achieve modularity in a consistent way throughout a whole company or product.

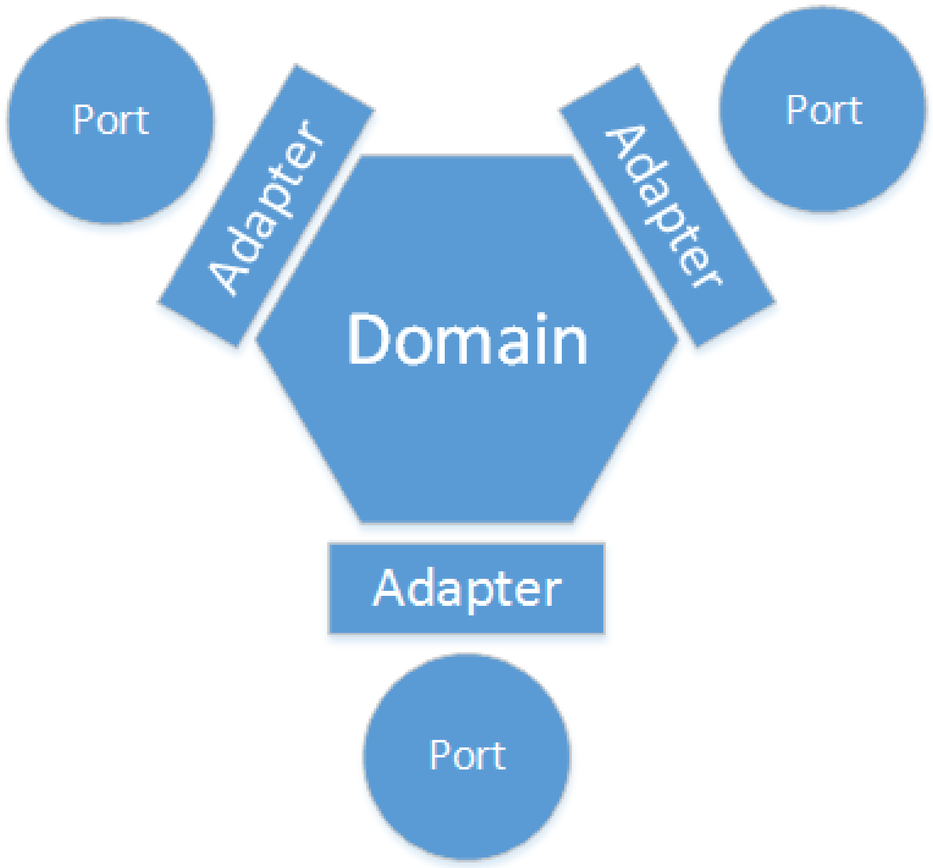


Figure 2.1.: The Ports and Adapters architecture pattern.

2.1. Knowledge Sharing in a Corporate Environment

Knowing best implementation practices for a certain technology can help during the development of software systems. But best practices only apply to a certain context or technology. Software development governance may also influence best practices so best practices can differ slightly between companies. Being able to share the experience from expert developers in a company as best practices with more inexperienced developer could increase the software quality. Documenting best practices and governance maybe necessary, but there is a gap between the documentation and how it can be applied. Newman proposed the idea of providing best practices and governance through code. This enables developer to run and explore code. The best way for sharing knowledge about best practices would be to see the code itself and have it explained by the expert developer. But this may not always be possible. But an exemplary application containing important best practices, available to every developer and developed by all would be a great start to share knowledge The Agile Manifesto favours working software over comprehensive documentation. So generating an exemplary software application, which follows the corporate guidelines and best practices instead of just documenting them, is a more agile approach for knowledge sharing. However this does not mean that no documentation is necessary any more. But being able to the experience the guidelines and best practices applied in the application example will probably help the software developer to implement them in their own software projects.

2.2. Motivation for a Code Generator

A modular architecture favouring separation of concern can decrease the complexity of the software system, but it may increase the complexity of its development environment. If new development environments have to be created on a regular basis this will become a big overhead. If a convention for code organization and a naming strategy is used this is a highly repetitive task. Code generators can help software developers and operationals to minimize the overhead to create the initial development environment. If a common code organization and naming strategy for an architecture exists, a generator can leverage those knowledge and provide the tooling for generating architecture components in the development environment. One or more architecture components are mapped to entities of the development environment which are often called projects by IDEs and build tools. In section 4.2 a more thorough definition of a project given.

While the amount of files to be created by a code generator may be biggest at the beginning it can still be useful later in the development process. Architecture patterns can contain components which can exists multiple times in a software system with slight variations. So even at a later time during the software development process a generator can be useful.

But also the generation of code in a project can be useful Either boilerplate code can be generated to fasten the software development or example code, which may use specific technologies and can be a guidance new developers [New15]. If corporate governance rule regarding to the implementation of specific functions shall be applied then the generated code following the rules can make the rules more visible for the software developer.

So a code generator, which knows the architecture pattern, its code organization, naming strategy and possible technologies for implementation of a software system, can make the initial development environment creation faster, can generated examples containing best practices as guidance for new developers and can generate code in compliance to corporate governance.

2.3. The Problem Statement

The main problem this thesis wants to address with a code generator is the fact, that development environments, which corresponds to the modular architecture of a software system, can be very complex, especially if they have multiple stakeholders. Dividing a software system into small services may result in multiple development environments, which each have to be created by a software development team and managed by them. From the main problem four sub-problems can be derived either directly or introduced by a code generator, which is not addressing the main problem in the right way. First of all creating development environments can be a repetitive task. A generator which can only create a new development environment but can not be used to extend one later on is wasting generation capabilities. Second software developers and operationals are stakeholder of the development environment. A generator, which supports only one stakeholder and is not usable by others due to technology limitations, again is wasting generation capabilities. In worst case to different generator have to be maintained doubling the amount of work. Third completely inexperienced developers and operationals may be overwhelmed by complex development environments. If they are just inexperienced with one technology they have to acquire the knowledge how to implement technologies in such an environment by themselves, while they may be several other implementations of the technology created by other developers. And fourth tools which are not build with ease of use in mind may be more likely rejected by developers and operationals.

2.4. Case Study at KISTERS AG

This thesis was accompanied by a case study at KISTERS AG. Some development teams at the KISTERS AG use the PAP to separate technology from the domain. A naming strategy, code organization and proposed technologies are defined for the PAP. A complete implementation of the PAP is called a slice. If slices become to big it can be separated into subslices, which ae again an implementation of the PAP.

The Maven archetype plug-in is a software extension for the build tool Maven. It facilitates the code generation by using so called archetypes. An archetype is a special Maven project containing the templates files and an XML based description of the generation process At KISTERS AG a Maven archetype is used to create the development environment of PAP software systems.

It is used to address the main problem of the problem statement in section 2.3, and even the third and fourth sub-problem but not the first and second one. First of all it does not address the main problem sufficiently because the PAP is not supported fully. Furthermore it can not be used incrementally. So at the beginning it may generate to much and afterwards it cannot be used any more. The current version of the PAP Maven archetype does not take the concerns of the operationals stakeholder sufficiently into account. an exemplary project can be generated for the inexperience stakeholders, but it can not be excluded for generation making the generator harder to use for experienced stakeholders because they have to delete unnecessary files first. Ease of use is sufficient for a command line tool, but providing just a command-line tool for generation may be not sufficient.

2.5. Requirements for a Code Generator

To better understand the requirements of a code generator, which are necessary to solve the problem stated above in section 2.3, interviews and an survey with employees at KISTERS AG were conducted. The questionnaire of the interviews is available in appendix A The same questions were used for the survey. Interviewed employees were architects, developer and tooling developer, especially maintainer of the PAP Maven archetype.

The first questions aimed to get a better understanding of the problem A. The others questions helped to specify the requirements for the code generator and were separated into 4 categories. First the interviewed person were ask about own ideas in general regarding improvements for code generation with the Maven archetype. Then they were asked specifically about ideas at the development phases design, coding and testing, about the tasks of setting up an development environment and refactoring code and about prototyping and examples. Then questions about basic ideas and generator functions were asked which could be important for the user of the generator. Additionally a list of possible generator features should be prioritized Afterwards questions about more advanced features, which may be expensive to implement and which usefulness was not validated until then, were asked. In particular these were questions about possible user interfaces, a model driven generation process allowing to update the generated code and about refactoring of architecture components. In the last part of the interviews question regarding the development and maintenance of the generator and templates were asked. By these questions ideas about separation of responsibilities within the generator, modelling of the architecture pattern and support for governance should be validated. At the end a feature list were presented which should be prioritized again. The questions regarding more specific code generator narrowed automatically the scope of the generator. The intention behind starting with open questions about own ideas was to gather new ideas which are not influenced by ideas from others.

From the interviews and the survey many ideas were gathered, some of them tended to be out of scope for a code generator, some were already implemented by other tools, which were developed at the same time the interviews took place. Most of the ideas can be found in the requirements. Stakeholders prioritized generator functions differently depending on their daily work. Software developers cared less about functions necessary to implement the generator but more about new technologies, which could be integrated as new templates into the generator.

All requirements were put in one of the following categories:

- Architecture Support (A) - Table 2.1: The generator has to fully support the architecture pattern including the naming strategy, code organization and supported technologies and infrastructure projects making it a heterogeneous development environment. Additionally the generator has to extendible to support future PAP implementation.

- Incremental Usage (I) - Table 2.2: The user shall be able to use the code generator incrementally at any time during the development. Additionally to an initial and complete generation of an example application a more fain grained generation process is necessary, allowing the user to select a specific component from PAP and adding it into an existing project.

- Ease of Use (U) - Table 2.3: The user shall be able to use the code generator as easy as possible. For an easy usage installation, updating and execution of the generator has to be considered.

- Knowledge Sharing (K) - Table 2.4: An generated exemplary application shall contain best practices to share those between developers. Developers shall be enable to participate in the development of the exemplary application.

2.6. Summary

Development environments of software system may be complex and time-consuming to created. If an architecture pattern is used, which provides a naming strategy and code organization, then a code generator can be used to create development environments of software system implementing the pattern.. Full architecture support, incremental usage, ease of use and knowledge sharing are considered to be important requirements for a code generator which can address the concerns of different stakeholder at different proficiency level.

Requirements for PAP architecture support	
Epic: The generator supports the PAP and used technologies now and in the future	
ID	Requirement
A.1	The code generator supports the naming strategy and code organization of an architecture
A.2	Heterogeneous code generation regarding the purpose of projects, like architecture components and infrastructure code
A.3	Generation of heterogeneous code regarding technologies like programming languages, build tools is possible
A.4	A simple way to extend the generator by new architecture components
A.5	A simple way to add new technologies influencing code organization within a project like build tools
A.6	A simple way to add architecture components implementing new technologies as exemplary code
A.7	The code generator can update itself automatically
A.8	The user can select a code generator version or otherwise the newest version is used
A.9	A code generation convention is easing the development new generators
A.10	A generation life cycle is easing the development of generators
A.11	The location of the generated content can be dynamically determined
A.12	A generation of a complete PAP slice is possible like the code generation performed by the PAP Maven archetype
A.13	Single PAP components can be chosen for generation
A.14	Full support of the PAP naming strategy and code organization
A.15	New PAP components, especially for infrastructure (Secondary JPA Adapter, arc42, Chef) are made available for code generation

Table 2.1.: Requirements for PAP architecture support.

Requirements for incremental code generation	
Epic: The code generator can be used incrementally	
ID	Requirement
I.1	New projects can be generated for architecture components and infrastructure code
I.2	New architecture components can be added into existing projects
I.3	Generation of example code is optional
I.4	Configurable project structure by selecting appropriate technologies for PAP components and infrastructure projects (similar to Spring initializer)
I.5	Composition of Generators of architecture components and infrastructure projects
I.6	Splitting and merging of architecture components on project level (Refactoring)

Table 2.2.: Requirements for incremental code generation.

Requirements regarding the ease of use	
Epic: The code generator is easy to use without prior knowledge about it	
ID	Requirement
U.1	The generator provides a textual user interface
U.2	The user can interactively input necessary information into the code generator
U.3	The user can pass all necessary information as arguments into the code generator
U.4	The generator can be used within IDEs by an IDE plugin (GUI)
U.5	File conflicts are detected by the generator and the user can resolve them
U.6	Possible projects for generation can be automatically detected by the code generator
U.7	Possible projects for generation are listed within a repository
U.8	Git is integrated into the code generator to create Git repositories for generated projects if requested

Table 2.3.: Requirements regarding the ease of use.

Requirements supporting the transfer of knowledge between software developers	
Epic: Software developers shall be able to share implementation best practices	
ID	Requirement
K.1	Code generation templates contain an exemplary application using best practices and governance
K.2	Software developers can easily reuse their code as templates for code generation
K.3	The templates files can be compiled
K.4	The implementation of the code generator and the templates shall be separated
K.5	The templates can be distributed into multiple small projects
K.6	The templates can be versioned
K.7	The templates can be build, tested and released
K.8	The version of a template used for code generation can be specified by the user
K.9	The user can select a template version or otherwise the latest template version is used

Table 2.4.: Requirements supporting the transfer of knowledge between software developers.

3. Related Work

Don't panic!

Douglas Adams

Contents

In this chapter existing generators and generator frameworks will be examined regarding the requirements of section 2.5. Generators can be categorized based on their properties. An important property of an generator is the way how the generated code shall be used. Before evaluating generators they shall be categorized based on this property. Then generator frameworks and generators allowing the needed usage of code are evaluated.

Herrington separates code generators into two categories. They can either be active taking long term responsibility over generated code, which means that changes to the generated should be avoided in most cases, or they can be passive, meaning the responsibility about the generated code is directly given to the user right after the generation [Her03]. The code generator, which will be presented in this thesis, has to be passive. Most code generator for model-driven software development are active and don't have to be included in the evaluation of code generators in this chapter.

3.1. Generator Frameworks

In this section a generator tool and framework will be presented, which can be used to build generators, especially scaffolding generators. Both are evaluated on their characteristics to be easily and incrementally usable Additionally the code generation capabilities are evaluated and if they are sufficient to fulfil the requirements for the architecture support.

3.1.1. Maven Archetype Plugin

The Maven archetype plug-in is a Maven plug-in, which is primarily used to generate new Maven projects based on project templates although there are no restrictions for any technology [Mav]. The project templates are called Maven archetypes. Every archetype contains a XML description file, which describes the information which have to be imputed by the user and which describes the files inside the template directory which shall be either just copied or processed by the Apache Velocity Template Engine.

Ease of use: Java and Maven have to be installed before the Maven archetype plug-in can be used. Once this done the plug-in can be directly executed without prior need for installation like every other Maven plug-in. At the beginning of the generation an archetype has to be chosen. It is possible to specify separate versions for the Maven archetype plug-in and the archetype itself. If no version is specified for the Maven archetype plug-in then the latest version is used automatically. But it is possible to specify every released plug-in version. The same is true for the archetype. Then the information specified in the XML file, which are necessary for the generation, have to be given. Everything piece of information is requested interactively but it is also possible pass the selected archetype and the further required information as arguments. The user has no possibility to manage file conflicts. If the base folder or the Project Object Model (POM) file, which shall be generated, already exists, then the generation process will abort. If a file already exists, the existing file is kept and the generation process continues. But overall the usage is very simple and encourages testing of archetypes.

Code Generation: The files used for generation can be either plain files or Apache Velocity template files and are bundle with a XML file describing the necessary user input and the files which shall be used for the generation. An archetype can contain either an example application or just the minimal set of files mandatory for all applications of this kind. That is completely up the the archetype developer. It is not possible to exclude files from generation based on user input. The Apache Velocity template engine is used internally to transform template files. It is only possible to change the content of files based on user input. Also a folder and file name convention exists which can be used to change those names. But it is not possible to modify the directory hierarchy in an other way. So the folder hierarchy is given by the template files.

There is special support for Maven as build tool. The user has to provide the group and artefact identifier. The developer of the archetype can rely on the fact that these information are always available. Additionally the Maven archetype plug-in handles modules of an reactor build, a feature to compose multiple Maven projects.

Incremental Generation: Due to the fact the even the existence of the base directory will abort the generation process, the Maven archetype plug-in can only be used to generate an initial Maven projects. Incremental usage is not possible as long Maven is used as build tool for the generated project. The Schema of the XML description for an archetype has a flag for partial generation but there is none documentation available how this works.

3.1.2. Yeoman

Yeoman is a generator framework written in JavaScript and requires NodeJS as runtime environment [Yeo]. A Yeoman generator bundles several generators into one executable unit. Every generator using Yeoman has one main generator and can have multiple sub-generators. The whole framework is very well documented. The public NPM repository is used to store generators, which are publicly available for everyone. The Yeoman website has an overview of all available generators.

Ease of use: The user has to install Yeoman manually via NPM, a NodeJS package manager, before any generator can be used. A generator has to be installed separately also via NPM before it can be used. Yeoman not just a generator framework but also a command line tool, which necessary to execute generators. Every update of Yeoman or a generator has to be done manually. Breaking changes within Yeoman would prevent generators from being executed until they are adapted. Yeoman provides an API to define a set of interactive user queries. The resulting user queries are well structured and easy to use for the generator user. Yeoman also provides convenient features for file conflict management. If the generator developer has used the proposed API for writing files then Yeoman will detect file conflicts automatically and displays the user the difference between two files in the console. The user can then decide which file to keep and the generator developer does not have to care about it.

Code Generation: Yeoman has a convention for organizing the code and templates within an generator. This eases the development of generators. A base generator class is provided which can be reused and which provides functions as hooks for extension. These functions are executed in a predefined order and yield a generation life cycle. This life cylce guides the developer when they are creating a generator. Within the predefined hooks normal JavaScript code can be written. Therefore Yeoman generators are very flexible and can be easily adopted. Processing od template files is done into an in-memory file system via an API first. This allows the Yeoman generator to handle the file conflict management. Yeoman uses the JavaScript template engine EJS which integrates well into Yeoman. The API for creating user prompts is easy to use by the generator developer.. Every Yeoman generator consist of one main generator and optionally sub-generators. Generators can make reuse of other generators.

Incremental Generation: A Yeoman generator can contain multiple sub generators. These can be used to only generate increments of an application while the main generator can reuse these sub generators. The composability of generators will allow reusability and therefore lessens the effort of maintenance. The file conflict management of Yeoman

Requirement	Maven Archetype Plugin	Yeoman
Easy installation	5	3
Easy/Automated updating	5	4
Interactive usage	4	5
Incremental usage	1	5
Code generation capabilities	2	5
Knowledge sharing via templates	2	2

Table 3.1.: Comparison of the Maven archetype plug-in and Yeoman regarding the requirements for a code generator.

can show the difference between two files and the user can chose between one of the files. It is not possible to merge two files.

3.1.3. Summary of Generator Framework Evaluation

To have a better overview of the evaluation the Maven archetype plug-in and the Yeoman generator framework are compared to each other and certain requirements. The rating can be between 1 and 5. A higher value means better support for requirement. The result is shown in detail in table 3.1.

The same perquisites as for the code generator presented in this thesis are assumed. It can be expected that Java and Maven are installed on the computer of the generator user. This favours the evaluation of the Maven archetype plug-in regarding an easy installation. Maven handles updates very well, getting the highest rating for easy and automated updates. Both provide good interactive usage support. The Maven archetype plug-in accepts every input either interactively or via arguments, while the interactive capabilities of Yeoman are really great. Regarding incremental usage Yeoman is much better ans also the code generation capabilities are very advanced. Both of them use template-based generation approach, but these templates are part of the code generator and cannot be accessed as easily as it is required for knowledge sharing.

3.2. Framework-specific Code Generators

In this section several generators for specific application frameworks are evaluated and compared regarding the requirements from section 2.5. These generators were developed for a specific framework and they contain a naming strategy and codeline organization for one or more architecture patterns, which can be used with the framework. It is not expected that these generators will support an arbitrary architecture pattern

3.2.1. Spring Roo

Spring is an enterprise application framework. Spring Roo is a command line shell to create Spring applications [RP12]. It integrates Maven as build tool.

Ease of use: Spring Roo provides very good documentation online and even a book [RP12]. Spring Roo is very powerful with a lot of options and is able to modify even existing files. All shell commands, which have been examined, could display a list of all possible arguments. The input for a command is always given as arguments and there is no interactive guidance. The shell provides auto completion for mandatory arguments in a predefined order without the necessity to start typing the argument name. Optional arguments have to be typed in partially before auto completion is possible. Next to the shell Spring Roo starts a web server when it is started. A few commands can be executed by a web interface. Additionally Spring Roo plug-ins for IntelliJ Idea and Eclipse exists.

Spring Roo add-ons contain the generator source code and the templates. These can be updated by a shell command. So the updating process is very ease for the user but it has to be done manually. Since Spring Roo is using OSGi, all add-ons are JAR files. It is possible to revert all updates going back to the installed version by deleting the newer JAR files. But it is not easily possible to use Spring Roo in different versions.

Via add-ons it is possible to integrate other tools. Spring Roo already has an add-on for Maven and for Git. The Maven add-on can be used to build the project by using the shell and the Git add-on can automatically commit changes triggered by other commands if the Git add-on is activated.

Incremental Generation: Depending on the state of the project the possibilities of code generation differ. First a new project has to be initialized based on project meta data. After that a very fine-grained and incremental usage of the tool is possible but a domain model is necessary. For example after executing a command, which adds the Spring MVC module to the project, new commands to add controllers or views or start the web application are available. The generated files contain configuration or just the boilerplate code necessary before logic can be added. The web add-on can also generate a scaffold for an entity creating the view, the controller and code for accessing the database, which are necessary for CRUD operations. Some of the commands does not trigger a generation but an adaptation of existing files, e.g. the logging level can be changed by a command which will not trigger a new generation of the logging property file but just an adaptation of the logging level.

Spring Roo can generate services, repositories, controllers and views. It works well if a domain model exists.

Knowledge Sharing and Governance via Code Generation: By using a domain model the generated files are useful for the problem domain, but they lack deeper logic beyond CRUD. Several Spring Roo commands can be combined into a script to generate an application for an exemplary domain model, but it is not possible to generate a complete application containing some more complex logic.

There is also no mechanism to extend existing add-ons. So if own best practices or governance shall be integrated, which differs from the one Spring Roo proposes, then existing add-ons have to be rewritten or new ones created. But it is possible to extend Spring Roo by add-ons. However, this requires deeper knowledge of the tool beyond its usage. Spring Roo uses templates and a model driven approach to alter Java code. But using the model driven approach is more complex then the template based. The templates or model transformations are part of the add-on and are released together. Whenever templates are modified the whole add-on has to be build and the resulting artefact has to be copied into the installation folder of Spring Roo before the modifications can be used. Some of the templates in the existing add-ons contain place holder and some use fixed strings.

Architecture support: Like the Spring framework itself, the core module of Spring Roo does not propose any architecture. But add-ons like the one for Spring MVC support the MVC or Active Record architecture pattern. As build tool only Maven can be used. Spring Roo only supports an application stored in a single Maven project. Mostly Java, JSP or XML files are generated, but it depends on the templates. AntLR is used to modify Java files.

To use different architecture patterns or technology Spring Roo has to be extend by add-ons. Core libraries, which eases the development of own code generators, are available though.

3.2.2. JHipster

JHipster is a generator for a web application using Spring as back end and AngularJS as front end technology [Jhi]. Additionally it provides further tooling and subsystems helpful to run the generated application.

In contrast most of the other generators presented in this chapter, JHipster can be used to generate a working application containing functionality which is commonly used by web applications like user registration and authentication, and application monitoring. JHipster supports many different technologies like databases or for caching, messaging and logging and also a few for the front end. It is also possible to choose between Maven and Gradle as build and dependency management tool. JHipster can also be used to deploy the application to supported cloud providers.

Since JHipster supports many technologies and the initial generated application contains already often used functionality its quite a huge application right from the beginning. Even configuration files for not used technologies are generated. This makes the whole project quite complex for beginners. JHipster may focus more on advanced users who are already familiar with the technology.

Ease of use: JHipster is based on Yeoman so it needs NodeJS as runtime environment. It can be installed via the Yarn or NPM package manager for NodeJS. Additionally a Vagrant managed virtual machine and a Docker container are available having JHipster installed. Since JHipster is based on Yeoman it can be used as a command line tool. Necessary input for JHipster can be done interactively. The user will be directed through the necessary information. This makes it easier for the user to directly execute JHipster without the need to study the help before. Except for one back end related architecture decision right at the beginning most of te other decisions are technology related. To build, test and run the generated application JHipster provides a GUI application. JHipster also offers a domain specific language (DSL) to model a domain. JHipster provides good documentation and supplementary tools to model the domain and use it as input for the generator. From the Yeoman generator framework JHipster also has the conflict management showing users the difference between conflicting files and letting them decide which to take.

Incremental Generation: Initially a Spring and AngularJS application can be generated by using the main generator. The generated application does not contain domain logic but it provides functionality for user authentication and an administration dashboard for monitoring the application. JHipster provides additional generators for incremental usage. The most complex of those is the so called entity generator. It alters the database and creates back end and front end code for CRUD functions. This generation is similar to the scaffolding from Spring Roo. If microservices are the preferred architecture then first a gateway application has to be created by using the main generator. Afterwards the main generator can be used incrementally to add more microservices to the overall application. For service discovery, configuration management and logging JHipster provides Docker container. The generated applications are configured to work with these containers.

Knowledge Sharing and Governance via Code Generation: JHipster generates not just a new project structure but already an application containing logic for registering users and to maintain the application via an dashboard. These are already best practices for Spring and AngularJS applications using common technologies. JHipster provides a possibility to extend its functionality by modules. These modules can be executed as standalone generator with access to project specific information provided by JHipster or they can be used as part of the main generator via hooks. JHipster uses the template based generation approach provided by the Yeoman generator framework.

Architecture support: JHipster supports two different back end architectures. The back end can be either monolithic or can be distributed into microservices. Other architectures are not supported. Due to the possibility to extend JHipster by modules it is possible to extend the architecture implementation by new technologies and integration into new build tools.

3.2.3. Ember CLI

Ember is a JavaScript framework for web applications [Emb]. Ember CLI is a command line tool for generating Ember projects. It uses a predefined set of technologies for building the application, as template engines, handling CSS files and more. Generation is done based on a code organization convention and a naming strategy. Ember CLI provides good documentation, both for its usage and extension. The naming convention and project structure are explained in the documentation.

Ease of use: Ember and Ember CLI both require the NodeJS JavaScript runtime environment. So Ember CLI users will have NodeJs installed because of Ember itself. The Ember CLI can only be used via its command line interface. It provides commands for the initial project creation, for subsequent code generation and for building, testing and running the application. Via a help command a description of every command and generator can be displayed in the console including the possible arguments. There is no code completion for CLI commands or the possibility to interactively input user information. So before executing a new command or generator the documentation has to be checked. There are many generators available which make the integrated help documentation hard to read. The number of generators can even increase with custom ones. Installing and updating Ember CLI can be done manually via NPM or any other NodeJS package manager. Every developer has to check them self if a new version is available. So it may happen that developers working on the same project are using different Ember CLI versions. Ember CLI detects file conflicts and can show the difference between two conflicting files. The user can decide which file to take.

Incremental Generation: The initial creation of a new project and subsequent code generation are separated into single commands. By the initial generation a new Ember project containing the complete project structure and a web application showing a welcome screen is generated. At this point it is possible to run the application to have a look at the welcome screen. While it does not contain any logic it demonstrates a running application. The subsequent generation is done by so called blueprints. Blueprints are code snippet generators for the Ember framework and architecture components. They are intended for an incremental usage of the Ember CLI. For example whenever a new Ember controller is needed it can be generated by a controller blueprint.

Knowledge Sharing and Governance via Code Generation: The initial project generation contains a very simple application showing a welcome screen. By extending Ember CLI with new generator add-ons it is possible create own generators which provide more value to software developers. These custom generators can be used for implementation governance or to show best practices. It is also possible to create a new generator which can generate an example application based on an argument. Every add-on has to be installed once per project. Afterwards it can be used by any developer. The version of a generator add-on is also set per project. Within an add-on multiple generators can be added. A project structure for generators exist as convention helping developers to build own generators. Ember CLI contains a command to generate a new add-on and new generators including folders for tests. This makes it easy to create new generators. A template based approach is used for generation. So the templates are very similar to the real code. Although they can not be tested right away because they are no real source code any more. Ember CLI provides hooks for developing add-ons and code generators. This hooks yield a generation life cycle which is a guidance for the development of generators. These hooks are meant for more complex tasks then just code generation. So they provide much flexibility but also add unnecessary complexity for simple tasks. Within these hooks it is possible to execute other Ember CLI commands or generators, which makes it possible to compose bigger generators by reusing smaller ones.

Architecture support: Out of the box Ember CLI only supports the architecture used by the Ember framework. With the Ember architecture at its core it additionally comes with a naming and code organization convention. For developing own generator add-ons Ember CLI provides resolver helping to generate files in compliance to the used naming strategy. Theoretically by extending Ember CLI with new generators a customized architecture, technologies, naming strategy and code organization can be implemented. But then the existing generators and resolvers can not be used any more and have to be reimplemented if they are still necessary.

3.2.4. Angular CLI

Angular 2 is a JavaScript single page application (SPA) web framework. Angular CLI is a command line tool able to generate the project structure and framework components necessary for an Angular 2 based web application. The Angular CLI is based on Ember CLI. So its usage and its capabilities for incremental generation, knowledge sharing, governance and architecture support are the same as with Ember CLI. But Angular CLI uses its own architecture, technologies, code organization and naming convention. Reusable generator features from Ember CLI are reused by the Angular CLI to come up with just the commands and generators necessary for and Angular 2 application.

3.2.5. Lightbend Activator

The Lightbend Activator is a tool for generating applications and building them using the sbt build tool [Lig]. Sbt, the acronym of simple build tool, is a build tool for Scala and Java applications. Lightbend Activator can be either used as command line application ar via a web user interface. Basically there are two different kind of generators within the Lightbend Activator. Either tutorials containing an example application can be generated or so called seed projects, which mostly just contain a build file including dependencies and a code organization. It is not possible to provide some kind of input like the application name so the generation is just a simple copying of files. Only the target folder can be specified. Every generator has to use sbt or otherwise Lightbend Activator will not be able to build, test or run the generated application. Since the files are just copied, the templates are the whole generator. A template is an sbt project, a tutorial HTML file, a license and some meta data. And the Lightbend Activator is a collection of multiple templates.

Ease of use: Lightbend Activator provides a web user interface (UI) which can be used to generate new applications and to build, test and run them. It is even possible to inspect the generated code within the browser. Created projects are remembered and can be opened The web ui can be used to select a template and generate a project. The templates can be filtered using some predefined tags. After a project is generated it can be build, tested or run from within the web UI or the generated folders and files can be inspected.

Incremental Generation: The seed applications are intended to generate the boilerplate code for a new project using a specific set of technology. Therefore they do not contain an example application or any kind of logic, but just the mandatory files, e.g. build files, source code or configuration. All examined generators have only generated a single project. An incremental usage is not intended by the Lightbend Activator. It would be possible to create multiple generators for the architecture component, but it could result in code duplication of the template files.

Knowledge Sharing and Governance via Code Generation: The example applications have the intention to help software developers get familiar with a specific set of technologies. They contain some explanation of the internal code organization and the implementation using these technologies. They are mostly simple applications demonstrating core functionalities of technologies and how they can be combined. So their purpose is sharing knowledge about the usage of technology, a good code organization and how to configure the sbt build tool for this particular example. The explanation of the tutorial is added as an HTML file and will be integrated into the web UI. Since the template projects are plain sbt projects with no further configuration they can be tested by just building the project. New templates can only be added globally as GitHub repository. There is no possibility to add private templates which are only accessible within an company. A web hook can be used for automatic updates when ever changes in the example application are added to the GitHub repository.

Architecture support: There is no special support for architectures. Since the template files are normal projects any architecture can be implemented. Although like said before there is no possibility to provide any input to change an architecture based on the needs of the user. So for every variation an own generator has to be created.

3.3. Summary

In this section several generator frameworks and generators were evaluated regarding the requirements form section 2.5.

Regarding the frameworks the Yeoman generator framework provides the necessary flexibility for code generation, but it requires its user to install the NodeJS runtime environment and it uses the NPM package manager. At KISTERS AG every software developer has the Java runtime environment and Maven installed but only a few NodeJS. Additionally KISTERS does not have an own NPM Repository, but only a Maven Repository. As an alternative NPM could be used to fetch packages from Git, but this makes the installation of generators more complicated. Generators are not updated automatically which may lead to an usage of outdated generators and slows down knowledge sharing and governance via code generation. And last there is no architecture and technology specific support for generator development which has to be developed either way.

The Lightbend Activator is the best fit for an easy knowledge sharing, but generators can only be added globally. All other evaluated generators support a specific framework, architecture pattern and a set of chosen technologies necessary to build and run applications using the framework. None of the generators fully matched the requirements for a code generator, but some of them had good support for incremental code generation and could easily be extended. Because the effort for adapting on of the generators and adding PAP support is still large and they all have some short comings for easy knowledge sharing and governance, none of them was chosen as basis for the generator developed in this thesis.

4. Code Generation Concepts for Operation-sensitive Development Environments

There are no facts,
only interpretations.

FRIEDRICH NIETZSCHE

Contents

The code generators presented in the last chapter could generate different parts of a software system. Most of them support the generation of an initial project including the proper configuration of a build tool and an organization of the source code files and other resources and they support the generation of architecture pattern components which is supported by the framework the generator was developed for.

To allow a better differentiation of the elements, which can be generated, two architecture view model are introduced and on of them is extended by missing models. This helps to understand the intention of the code generator developed for this thesis and at which point the code generator has to provide additional support to enable the generation of needed architecture pattern components and technologies.

4.1. Background: Architecture Views and Viewpoints

Modelling the whole architecture of a software system into view will make the model hard to read and to maintain [RW11]. ISO 42010, a standard for a software system architecture description, defines views as a representation of those parts of a software system architecture, which are related to a stakeholder concern [Iso]. A stakeholder is defined by ISO 42010 as a team, individual or organization having an interest in a system and a concern is defined as an interest in a system relevant to one or more stakeholders [Iso]. To make it easier for software architects and stakeholders to create and understand views, ISO 42010 defines the concept of viewpoints as conventions for those tasks. The generated code has to address multiple concerns of different stakeholders. The ISO 42010 defines only a standard for describing views and viewpoints but no viewpoints itself. In this section two viewpoint models are introduced which will be used later in this chapter to map stakeholders, their concerns and generated code to those viewpoints.

4.1.1. The 4+1 Architectural by Kruchten

In 1995 Kruchten proposed a viewpoint model of software architecture to represent architectural structure for common stakeholder concerns. At this time the concept of viewpoints and the distinction between views and viewpoints did not exists explicitly and the term view was used for views and viewpoints. The model consists of five components and is shown in figure 4.1. For each view an own notation was proposed. At that time UML was not widely known and did not became a standard until 1997 [Col+97]. In 2007 a white paper was published which proposed a mapping of UML 2 diagrams to the views presented by Kruchten [Muc07]. In the centre of the architectural model are the scenarios or nowadays mostly called use cases. The four views are separated into conceptual views, which are the logical and the process view, and into physical views, which are the development and the deployment/physical view. Stakeholders and their concerns are shown in figure 4.1 for each of the four views.

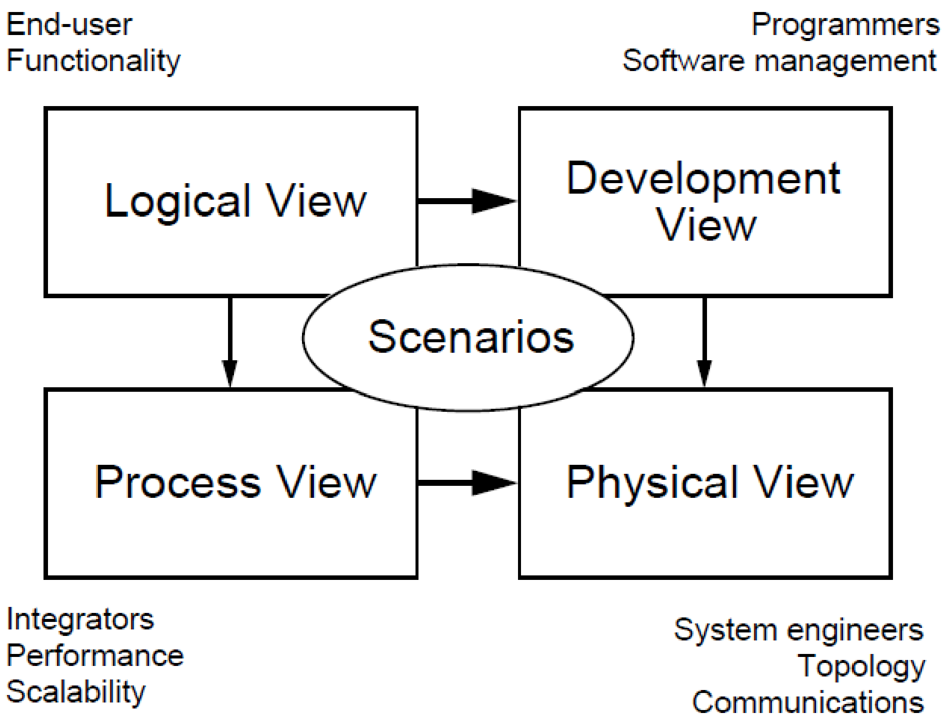


Figure 4.1.: The 4+1 architecture view model [Kru95].

The logical view is a structural model of the problem domain supporting the functional requirements [Kru95]. Structural diagrams and notations used for this view should support abstraction and decomposition. For domain centric architectures the domain model would be part of the logical view. Because the problem domains are very different for every software project the generator cannot easily provide domain specific support without being very specific about the used technologies. It can however generate components of an architecture pattern used for modelling this view with an exemplary problem domain to show best practices and tools useful for the implementation. But this code would not match the uses cases and has to be deleted shortly after partially or completely. IDEs and other programming language specific tools provide good support for generating packages and classes to implement the model of the logical view. UML 2 class, object, package, state and composite structure diagrams would be a good fit for modelling the logical view [Muc07]. So the logical view is about modelling the problem domain.

In the process view the execution of tasks and their collaboration are modelled [Kru95]. Tasks are the operation of the main abstractions modelled in the logical view and are grouped into processes. Within the process view non-functional requirements and aspects like performance, concurrency, and distribution can be modelled. Modelling within this can happen on different levels. On the highest level processes are used to model a network running across hardware. Like the process view this one also depends on requirements which are very different for various projects. The UML 2 sequence, communication, activity, timing and interaction diagrams can be used for modelling the process view [Muc07]. So the process view is about mapping entities from the logical view to executable units.

The development view focuses on the software module organization within the development environment [Kru95]. Here software modules are the entities of the development view and are organized based on their physical packaging as deployment units. Here modelling is done at the physical level in contrast to the logical view, which is at the conceptual level. Within the development view requirements which ease the development and deployment are taken into account. In contrast to the other views, this one only indirectly influenced by the use cases and more about the concerns of the software developers. The UML 2 component diagrams can be used for modelling the development view [Muc07]. So the development is about composing entities from the logical view into software modules and mapping those to physical deployment units.

In the physical view the execution of software on computers, called nodes, is modelled [Kru95]. The physical view, which is sometimes also called deployment view, focuses on the hardware topology, its provisioning and how the components are distributed. Similar to the process view non-functional requirements and aspects of systems like availability, reliability, performance and scalability are considered. Infrastructure as Code (IaC) is an approach which treats infrastructure as source code and applies best practices from software development [Mor15]. IaC enables the automation of provisioning and deployment. It can be used to implement a runtime platform model. With provisioning tools like Chef, Puppet, Ansible or Salt, which support infrastructure as code, the distribution and provisioning can be put into source code and becomes part of the development environment. Because these requirements differ between software projects the nodes, processes, tasks and deployment units, which are represented in this view, can not be generated. But it is possible to generate the boilerplate code of technologies, like

Chef, which are used to implement this view. In UML2 Deployment Diagrams can be used to model the physical view [Muc07]. So the physical view is about mapping the software as executable units from the process view to physical nodes.

The logical, process and physical view take requirements related to product as outcome of a software project into account. The development view takes requirements from the development process into account which does not, or only little, emerge from the software system itself but more from the organisation, the chosen architecture and technology, and the people involved in the software development process, especially software architects, developers and testers. An organisation, which reuses an architecture pattern and associated technologies throughout multiple software projects, can benefit from a code generator, which enables the generation of software modules, infrastructure code modules and other shared physical modules and takes care of the organization of these modules in the development environment.

4.1.2. The Architectural View Model by Rozanski and Woods

Rozanski and Woods developed another view model which is extending the one by Kruchten [RW11]. In addition it uses the definition of views and viewpoints from the ISO 42010 standard. So in this section and in all following the term viewpoint will be used according to the definition to replace the term view and how it was used in the last section. Views are structures a software system, which use the conventions provided by a viewpoint.

The view model by Rozanski and Woods contains seven viewpoints, which can be used to create and understand the views of a software system. It is shown in figure 4.2. The intentions and responsibilities of views are well described within the viewpoints. The first viewpoint is the context viewpoint which defines interactions between the system and its environment. Furthermore the view model contains the functional, information and concurrency viewpoints, which are grouped together. Functional and concurrency viewpoints are similar to the logical and process viewpoints by Kruchten. The information viewpoint is a new convention for a data-centric view of a software system. The next viewpoint is the development view, which is basically the same as in the model by Kruchten. The last two viewpoints are the deployment and the operational viewpoint, which are also grouped together. Those two separate the concerns of deploying a software system and running it.

Not all viewpoints are important for the code generator presented by this thesis. So at this point only the functional, development, and deployment viewpoint will be explained in more detail. It is not ruled out that code generation is not possible for the other viewpoints but the code generator will not explicitly support it. Rozanski and Woods defined several stakeholders. Such a detailed distinction of stakeholders is not necessary fo this thesis. So only the ones, which are important for the code generator, will be mentioned explicitly.

The functional viewpoint contains the functional runtime elements [RW11]. For each of those their responsibility and interfaces are described and their interactions among each other. Often it is the core architecture structure and influences others like the

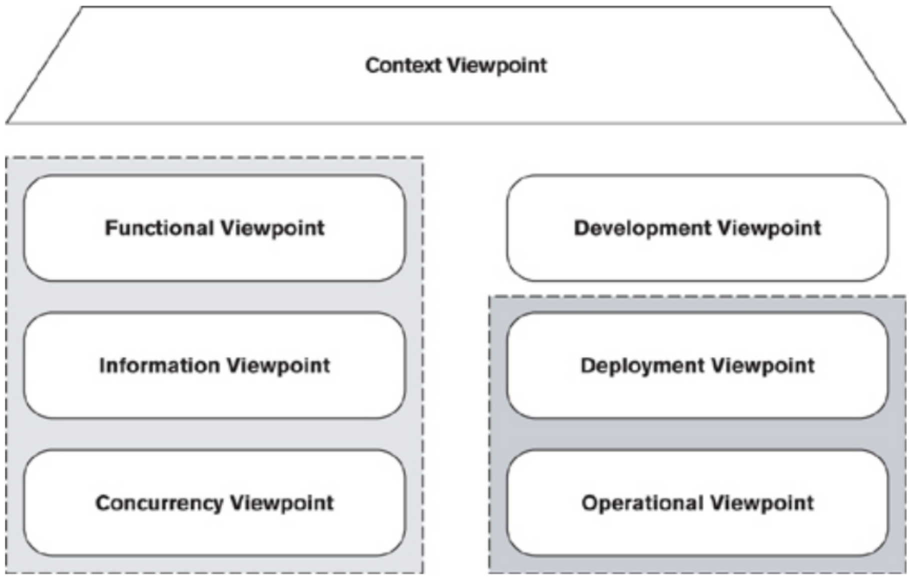


Figure 4.2.: Viewpoint Groupings [RW11].

information, concurrency or deployment structure. The elements of a functional structure model are the the functional runtime elements, their interfaces, connectors and external entities. The UML component diagram is mentioned as a good notation for this view. Architecture patterns can be used as template for the structure of a functional view. Design characteristics are coupling, cohesion, extensibility among others. The functional view is important to every stakeholder. The code generator shall generate an exemplary application to illustrate best practices for software development and implementation of preferred technologies. This makes the functional viewpoint important for the code generator. But the functional view from the exemplary application is a different one as the one from the software system, which the developer has to implement. Therefore the generated content belonging to this view has to be deleted, partial or complete.

The development viewpoint contains a convention for structuring the development environment, which includes the organization of source code files into modules to build, test and release them. Different models are part of the development viewpoint. The module structure model can be used to organize source code files into modules. Those modules can be grouped into layers to manage the dependencies between them. The UML component diagram with packages is a good notation for this model. Codeline organization is defined as the directory structure of source code, which is managed by a configuration management system (CVS), with the intention to build, test, and release the source code as binaries. Rozanski and Woods did not suggest a particular notation for the codeline model because in most cases using text, tables, and a few clear diagrams to explain the convention, how the source code files should be organized, should suffice. Software developers, testers, and product engineers, who are responsible for the deployment of the software system, are the stakeholders of the development view. The components of a software architecture pattern can be used as modules in the module structure model. The naming strategy and code organization of an architecture pattern can be used as codeline organization. These two facts make the development viewpoint very important for the code generator presented in this thesis.

The deployment view is a model of the environment into which the resulting binaries of the system, which are modelled by the development view, are deployed. The core model is called the runtime platform model and contains hardware nodes for processing, storage, and clients and network links between those. Additionally the functional elements are mapped to processing nodes. UML deployment diagrams can be used as a notation for this model. If infrastructure becomes code, it has to be managed by a CVS and becomes therefore part of the development environment. The code generator can generate exemplary IaC files, but the same as said about the functional viewpoint applies here.

4.2. Software Project Model

Rozanski and Woods did not define an explicit model using a domain language for the development viewpoint. In this section a model of the development environment is presented, which can be used as basis for a codeline organization. Additionally IaC makes the operational also a stakeholder of the development viewpoint.

The codeline model in the development viewpoint has the task to show the organization of the source code files. In particular how it should be build, tested and released and which tools shall be used for this purpose. Text, tables and simple diagrams are assumed to be sufficient in most cases for this model. The code generator has to generate code on accordance to the codeline model. A language and defined terms of the content, which has to be generated, ease the communication between software developer and code generator. Therefore in this section a model and its elements are defined, which can be used as a notation for a codeline model.

The development environment of a software system contains source code files, build scripts, documentation, and other files like images. These files can be organized in directories, which are a hierarchical structure. Build tools like Maven or Gradle, or IDEs like Eclipse, IntelliJ Idea and Visual Studio allow the decomposition of the development environment of a software system into so-called projects. The term project is often used for the development environment of a software system or its decomposed parts, but there is no consistent definition. Here a project is a entity to organize the development environment hierarchically, which is shown in figure 4.3. Every development environment starts with a root project, which can contain other sub-projects. A project can contain meta information about the project. In most cases it also contains a build script, which will be used to build, test, and release the source code files of the project and sub-projects. On developer computers a project is represented physically by a directory. The organization of the project in directories can map the hierarchical order of the projects but it doesn't has to. This definition of a project can also be mapped to the organization in cloud development environments like the ones provided by Eclipse Che [Ecl].

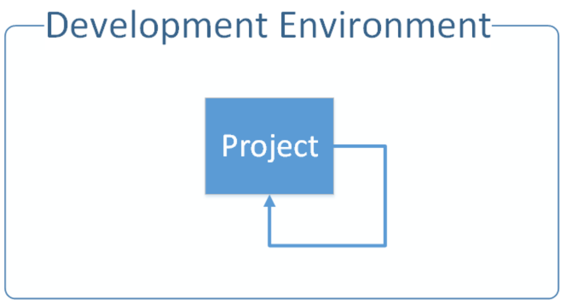


Figure 4.3.: The project model.

Based on this definition of a project and its hierarchical composition a software project model was developed to achieve a better understanding of the different project types, which can exists in a development environment. This model shown in figure 4.4 and contains common types of projects and defines the relationships between them. The root project is called the software system project and can contain other types of projects. First there are the software projects containing one or multiple modules from the module structure viewpoint. The number of software projects depends on the preferences of the software developers and can be even distinct for the same architecture pattern, which defines modules. The next kind of projects are the software deployment projects. Projects building, testing, and releasing a software, which can be executed as a single process, belong to this kind. A integration test project which is testing the interaction between the software modules but mocking any other system is an example for a software deployment project. System deployment projects are about system-wide deployment and testing. Acceptance and performance test projects but also IaC projects like chef recipes are software deployment projects. Finally there are documentation and build projects which are grouped together as supplement projects in 4.4. A build pipeline project or a project providing common build configuration are two possibilities for supplement project related to software build concerns. The software system project can contain an arbitrary number of the sub-projects. Between those sub-projects certain relationships are common. Software deployment project always depend on the software modules or other software deployment project. System deployment project always depends on one or more software deployment project. Those dependencies are illustrated in 4.4. Only dependencies, which could always be observed in an exemplary software system were added to the model. So the absence of dependencies in the model does not mean that there may not be dependencies between two projects. If a software system project contains a project for a common build tool configuration than every project containing the using the same build tool may depend on this supplement project.

Regarding viewpoints software projects are related to the functional viewpoint, software deployment projects to the development viewpoint and system deployment projects to the deployment viewpoint. All these viewpoints have different stakeholders. So knowing the kind of project, which shall be generated, also helps to find the stakeholder, which

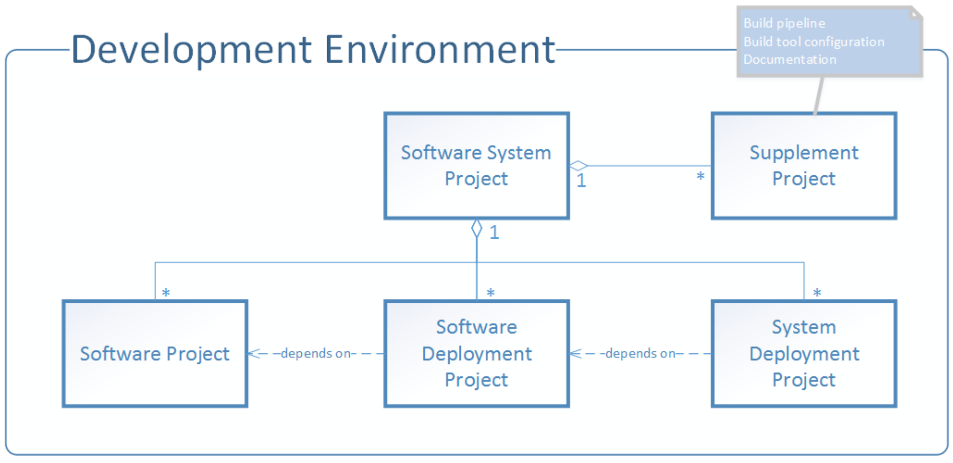


Figure 4.4.: The software project model.

benefits most likely from the generator functionality to generate such a project.

4.3. Stakeholders

There are multiple stakeholders, which are having different concerns for the content generated by the code generator. These concerns influence the functionality of different modules of the code generator. To better understand the intention of the generator concepts the stakeholders and their concerns will be explained first. The generator has three main stakeholders. These are the software developer, the software architect, the operational and the generator developer. The software developer has to implement functional requirements following an architecture and wants to use the generator for the daily development work throughout the whole software development phase. The operational has to implement infrastructure projects automating the provisioning and operation of the software system. Both want the generator to just generate needed code and nothing more. The generator has be easy to use without prior knowledge. Updates of the generator and the templates shall be as easy as possible. The software developer and operational shall be able to make own modifications of the templates and directly use them. If the modification may have value for others there has to be a possibility to easily share them with others. So mainly from section 2.5 the requirements about ease of use, incremental code generation and knowledge sharing are important for the software developer and the operational. While both have the same concerns different viewpoints are important to them resulting in a heterogeneous development environment, not only regarding used technologies but also stakeholder concerns.

The software architect wants the developers to generate code according to the specific architecture of a software project following an architecture pattern. The code generator has to be able to generate code which implements the chosen technology and follow a naming strategy and codeline organization of the architecture pattern. Therefore a fine-grained selection of the generated code has to be available. If technologies are missing and they are needed more often than in just one project, it shall be easy to add this technology to the existing generator as a selectable option for code generation. Modifications of the templates or the generator shall be used by the developer as fast as possible. So from section 2.5 the requirements about incremental code generation and architecture and technology support are important for the software architect.

The generator developer wants to easily extend and adapt the generator if necessary according to new requirements for the generator but also to integrate corporate-wide used architecture patterns and technologies. Adaptations to the templates by other shall be easy to integrate. So from section 2.5 the requirements about generator adaptation and the support for an architecture pattern and the needed technology are important for the generator developer.

These four stakeholders can be mapped to the viewpoints described in section 4.1.2. The architect has to model the necessary views based on the viewpoints. These views needs to be implemented by the developer or operational in accordance to the development view. For the software developer the functional, information and concurrency view point are important. The generated code will be used to implement the elements in the functional view, interacting in accordance to the concurrency view and following the flow of information from the informational view. For the operational the deployment and the operational view are important. The generated code and projects will be used to implement the distribution of deployment units in accordance to the nodes in the deployment view. The code generated by the software developer or operational has to be organized in projects in compliance to the development view. The generator developer has to extend the generator if it does not support an architecture element within the development view or an technology added by the architect.

4.4. Summary

Different stakeholders and with different concerns will use the code generator. Viewpoints refer to those concerns and help to understand, what problems the generated content has to address. For the functional viewpoint and deployment viewpoint exemplary and boilerplate code has to be generated. Those code has to be organized in the development environment, which can be very heterogeneous because modules from two different viewpoints with different concerns are part of it.. The software project model presented in this chapter helps to organize and understand the development environment. This knowledge can be used to come up with an architecture and concepts for a code generator.

5. Code Generator Architecture and Concepts

There are no facts,
only interpretations.

Friedrich Nietzsche

Contents

In this chapter concepts for an incremental and heterogeneous code generator and its architecture to implement the requirements stated in section 2.5 will be presented. The intention of the generator is to enable software developers and operationals to incrementally generate a development environment matching the architecture and requirements of a software system. Optional an exemplary application shall be generated into the development environment helping to share best practices and governance for implementing particular technologies among software developers. The expected benefit is to fasten the development of software system and to share knowledge and governance about the implementation of technologies between developers. Ease of use, incremental code generation, architecture and technology support and knowledge sharing are the four principles which are behind the concepts and architecture.

The primary user of the code generator is the software developer and the operationals. Most concepts presented in this chapter intend to make the result of the code generator directly or indirectly more valuable to the developer. The result of the code generator will be generated source code files, configuration files and whole software projects. The generator shall be made more valuable by having a wide range of well maintained and useful templates and by enabling the user to generated only the parts needed for the current task. The value of the code generator for developers strongly depends on the the quality and the usefulness of templates. If code is generated which does not work or does not work the way it is expected, the developer will not trust the content generated from templates. A good way to determine if it works is to compile, test and run the generated content. Content created by templates is useful for the developer if it can be generated in the right place and if the content uses the right technology which can be used to solve the current problem. So creating new templates, maintaining them and integrating them in the generator framework should be as easy as possible.

First the generator architecture and its modules are presented. Afterwards stakeholders are mapped to roles, which are related to the development and usage of the code generator. This shall help to better understand the scope of the concepts present here and its main beneficiaries. Afterwards the single layers from the generator architecture are presented from bottom to top

5.1. The Generator Architecture

In this section the architecture of the code generator will be explained. Projects containing architecture components and infrastructure code have to be generated to fasten software development. Therefore a set of technologies has to be selected, which shall be used to implement the architecture and infrastructure. Neither the architecture nor the technology is fixed and my vary for different software system, which results in a heterogeneous development environment. Another goal is the generation of an exemplary application. To implement the functionality of this application another set of technologies has to be used. This technology will be part of the templates and not the generator and is limited to a single project. This means that the generated technology implementation is not affected by the occurrence of other projects and does not affect them. So code generation happens on two different levels for two different purposes.

For the code generator a layered architecture was chosen. Layers encapsulate a set of coherent functionality and minimize the dependencies between them [Fow02]. Different stakeholders will participate at the development of the code generator. The layers shall also give them guidance which parts of the code generator they want to extend. The code generator is separated into four layers. Each has its own purpose and addresses the concerns of a particular stakeholder. An upper layer may have dependencies to a lower one but not the other way around Figure 5.1 shows the four layers.

Frameworks are incomplete applications providing interfaces to customize them to ones needs [FS97]. Architecture and an exemplary applications are considered to be customizations of this code generator. The bottom layer is called the generator framework layer containing the functionality for developing code generators matching the requirements from section 2.5. APIs and a code generation life cycle will be provided by this layer. This layer is completely independent from the architecture patterns and technologies, which may be used in the generated code.

The second layer is called the architecture and technology layer and has two purposes. On the one hand it contains the naming strategy and a convention for codeline organization of architecture patterns. On the other hands it contains common functions necessary to deal with technologies which influence the codeline organization or are influenced by

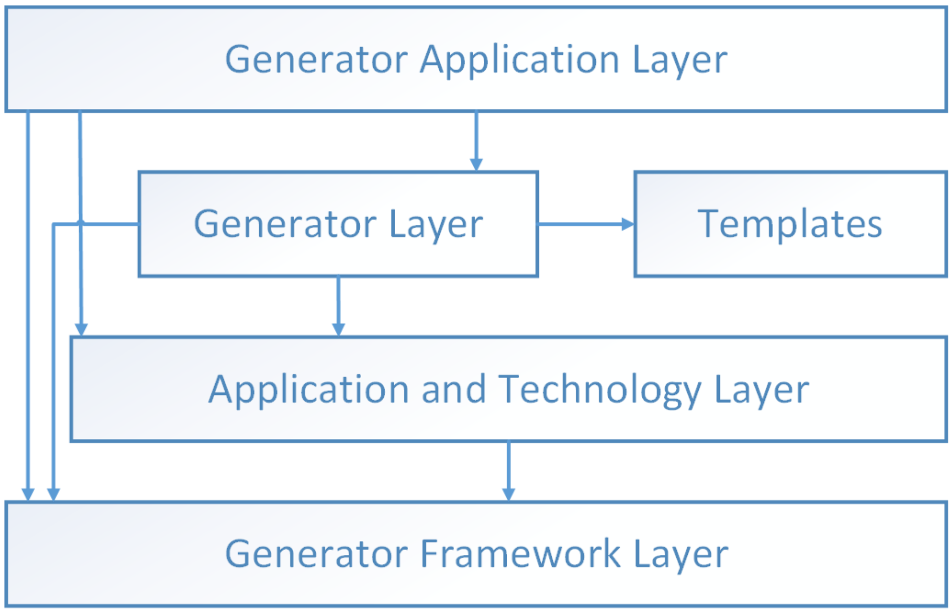

Generator Application Layer
Generator Layer
Templates
Application and Technology Layer
Generator Framework Layer

Figure 5.1.: The four layered generator architecture.

it. An example would be an API to modify Maven POM files based on the fact which architecture components are present in the development environment.

The third layer is called the generator layer. It contains the actual generator implementation and the templates. Within the context of this code generator a generator is defined as a executable piece of code, which is used to generate content and which follows the generator life cylce provided by the generator framework. Another thing to mention is the fact that templates will be stored separated from the generators which is requested by requirement K.4. This is shown in figure 5.1 which illustrates that the templates and generators are separate entities.

And the fourth and last layer is the generator application layer. In contrast to a single generator, several generators are bundled together and provided as a code generator application. According to requirement A.7 a code generator application has to be able to update itself. The generator application will be a wrapper for generators providing this capability.

Figure 5.2 shows a coarse view of the architecture of an generator application. This application contains three generators as an example. Depending on the complexity of the architecture pattern this can be much more. In this example every generator has its own template files for the generation process. APIs providing a naming strategy and codeline organization of an architecture pattern and supporting two technologies are available. Common generator functions are usable by the generator framework.

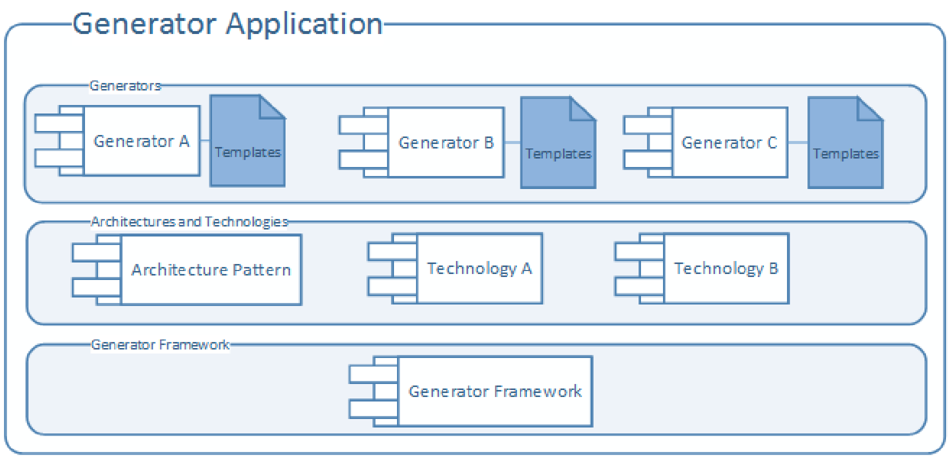


Figure 5.2.: An overview of the modules contained in an exemplary generator application.

5.2. Generator Roles

Every of the four layers defined in section section 5.1 has to be developed by some one and will be used by some one. This categorization of interactions with the generator is a coarse one but it helps to keep in mind who is using the modules of a layer. The stakeholders of the generator are software architects, software developers, operationals and tooling developers and are described in section 4.3. These people within a company have concerns related to the code generator and will either use or develop the code generator. Software developers shall use the generator but they shall also participate in the development of the code generator templates as a mechanism to distribute their knowledge among other software developers. But software developers will not be the only ones developing the generator. So on the one hand stakeholders will perform different tasks like using or developing certain architecture layers of the generator and on the other hand one task regarding a particular architecture layer can be performed by different stakeholders.

In this section generator roles will be introduced to map stakeholders to architecture layers by the task of either using or developing this layer. Every role will develop modules from at most one layer but may use components from multiple other layers. The usage of modules from other layers corresponds to the relationship between these layers from figure 5.1. The purpose of the roles is to ease the detailed explanation of the layers in the next section regarding the generator related concerns of the stakeholders. The categorization of stakeholders into roles based on their interaction with modules from a layer is very coarse but it will help to better understand who is using a module. So these roles map different stakeholders to an architecture layers by the two tasks of developing and using generator modules. In total there are five different roles, which are the generator framework developer, the architecture and technology developer, the generator developer, the generator application developer and the generator user. The mapping is shown in

5.3. Except for the generator core function developer and the generator user every role is developing modules for one architecture layer and may use modules from layers below.

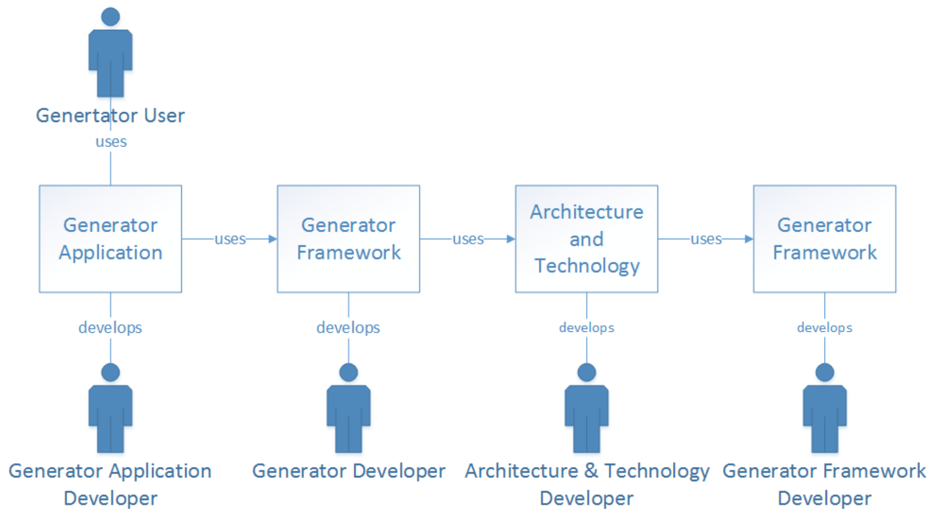


Figure 5.3.: Mapping of roles to generator architecture layers.

The first role is the generator framework developer, who has to develop the core code generation functionality. Only the tooling developer will be acting as this role to develop APIs for common code generation functions which can be used by modules from the other three layers. The architecture and technology developer, the generator developer and the generator application developer are possible users of the modules developed by the generator framework developer.

The next role is the architecture and technology developer who shall provide an API for naming strategies and codeline organization of architecture patterns. Additional APIs for technologies which are affected by codeline organization and naming of generated content has to be developed by this role too. The software architect will act as this role to provide the architecture and technology related APIs. To achieve a common user experience for different architecture patterns the tooling developer should support the software architect.

The next role is the generator developer who will develop specific generator in the generator layer. Software developer, software architects, operationals and tooling developers are stakeholders which are intended to be a generator developer. The development of generators is intended to be done collaboratively and continuously over time by different stakeholders and not just by one. The tooling developer will support the other stakeholders with knowledge about the use of the generator framework either directly or via documentation. Software architects, developers and operationals will implement generators which are useful to them. The development of a generator can be separated into developing templates first and then writing the generator code itself.

The purpose of the generator application developer role is to provide a selection of developed generator to a team of software developers, which are useful for them.

Generators will be bundled in the application layer and not every generator will be useful to every software developer st any time. Based on the architecture and technology, which shall be used to develop a software system, the software architect has to decide which generators shall be bundle into a generator application and mkae them available for the generator user role.

The last role is called the generator user. Stakeholders in this role will use the generator tool to generate content like source code based on their needs. In most cases the software developer will be the stakeholders acting as a generator user. But for testing the generator and its templates other stakeholders will be a generator user too.

Since responsibilities of each role are known the four layers can be explained in more detail now.

5.3. Generator Framework Layer

A framework offers generic solutions to similar problems within a specific context [Zü05]. According to Fayad and Schmidt they are semi-complete and provide stable interfaces and hook methods which allow applications to extend these interfaces. The generator framework layer shall be a framework for code generation. It provide interfaces for common functions which are necessary for the code generation but also an control flow of the code generation process. Hook methods integrated in this control flow can be extend by generators in the generator layer. The generator framework layer is the bottom layer and will be developed by the tooling developer as the generator framework developer role. It can be used by any layer above. First the generation life cycle will presented and afterwards two APIs which are very useful in certain phases of the generation life cycle.

5.3.1. Generation Life cycle

The generation process can be separated into three steps and is shown in 5.4. The first step is called the configuring phase. Within this phase the creation of the model necessary for the generation shall be accomplished. Therefore arguments passed to the generator are evaluated, the user is asked for missing information and finally the model for the generation is build. In the second phase, which is called the writing phase, the templates are processed and files are copied. In the last phase the generated files can be adapted based on other generated files and projects. Initialisation and clean up is handled by the generator framework. This phase is called the finalizing phase.

5.3.2. Composable Generators

To achieve reusability of generators for an incremental usage they have to be composable. In particular a generator which can generate the whole domain layer of the PA pattern should reuse generators which can only generate the API, SPI and implementation. Still a composable generator should have the same life cycle like an atomic one. This means that a composable generator has to execute in a certain phase all phase of the same type of all its sub-generators before it can move on to the next phase. Otherwise a generator

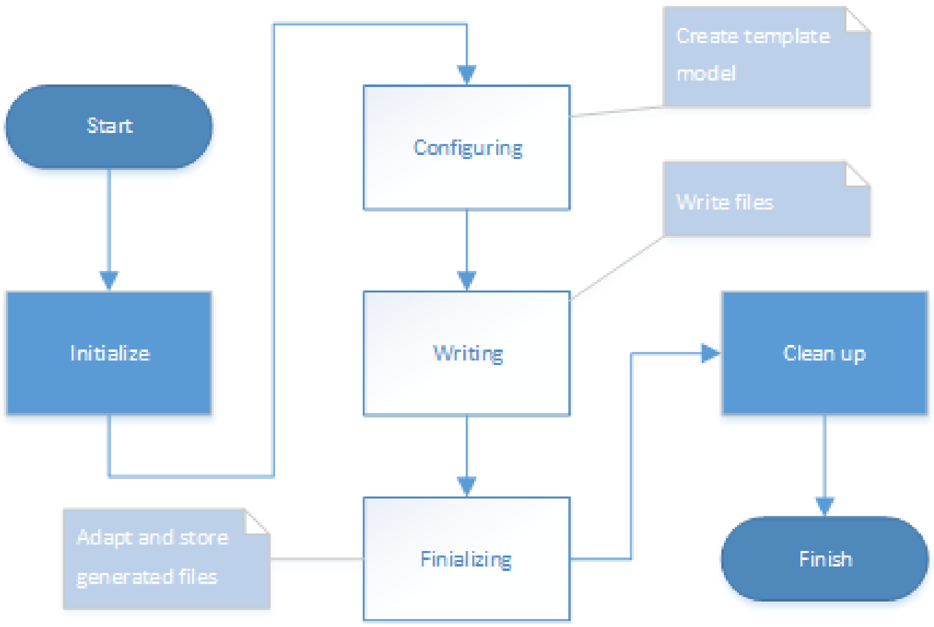


Figure 5.4.: Three-phased Generation Workflow

could not know which other projects are generated and act accordingly if adaptation would be necessary.

5.3.3. User Query API

To build the template model in the configuring phase it may be necessary that the generator user has to provide information. Therefore the generator framework shall provide an API, supporting the generator developer to ask the generator user for input. In this context a query is a question for the generator user associated with a keyword to retrieve the answer once the question is answered.

The Yeoman generator framework provides access to an such an API [Yeo]. Four types of user queries which are also support by Yeoman seem in particular useful. The first user query type will prompt the user to input arbitrary text related to a question. This kind of query will be called input query. The second will prompt the user to confirm a question with yes or no. This kind of query is called confirmation query. The third query will prompt the user to choose one answer out of a list of many. It not only has to provide a question but also a list with possible answers to the user. This user query is called a select query. The last query will prompt the user to select multiple possible answer out of many. This is very similar to the third user query and called a multi-select query.

Often not only one question but a many needs to be answered by the user. So a sequence of user queries has to be created by the generator developer. If an architecture has multiple components and for each one of them an own generator shall be created than they may need common information related to the architecture pattern. Having every generator to implement the same user query again increase the maintenance effort of the generators and can lead to worse user experience for the generator user because the same question is asked in different ways. Therefore a sequence of user queries has to be shareable and composable. For example a generator developer may want to combine own, custom user queries with a sequence of common user queries related to an architecture pattern and provided the architecture API. This will also require that a user query may not be executed right away but can be explicitly executed at a later time by the generator developer.

In case another user query is handed over it may be possible that the questions have already been answered. So when a user query is executed it has to be possible to hand over answers, which can be mapped to questions by the keyword. So before a question is asked, it has to be checked first, that it was not answered already. Otherwise it has to be skipped.

Some questions may lead to further questions depending on the answer of the user. Therefore for some sequences of questions it would be useful to build up a hierarchy of questions. For example the user query to confirm a question can be used to ask different questions based on the answer of the user, whether it is yes or no.

The concepts of an user query API presented above should result in an API which eases the task for the generator developer of getting the necessary input from generator user.

5.3.4. Intermediate File Storage

Processing template files and storing them on the disk is the main task of the writing phase from the code generation life cycle. Often several files shall be generated together which depend on each other and can not be used independently form the other files. Therefore the concept of an intermediate file storage (IFS) is proposed. The file storage is called intermediate because from a file process point of view it is storing and processing the files only temporarily until they are provided to the user.

The IFS shall manage the retrieval of all files necessary for an execution of a generator. Additionally it will be responsible for processing the templates files into the expected output. Before a file can be written onto disk as expected several steps have to be performed which are shown in figure 5.5. The template files need to be loaded from the template location, transformed into the expected content and then placed onto disk. During the process several things can go wrong and have to be handled appropriately by the IFS. It shall be avoided that only some of the files are written onto disk due to an error which happened during code generation. Therefore all necessary files have to be be loaded and transformed first. It is up to the implementation if all files are loaded first and then transformed or if every loaded file is transformed first before the next file is loaded. The result has to be stored at a temporary location, which can be for example on disk or in memory. Only after every required template file has been processed, all files shall be written to the expected output. To execute the writing to the expected output the IFS has to provide a function for the generator developer which shall be used if no files need to be added any more. Optionally before writing the files in the finalizing phase they could be transformed based on generated projects and architecture components. Because everything was written in an IFS, it is easier to make changes in the finalizing phase.

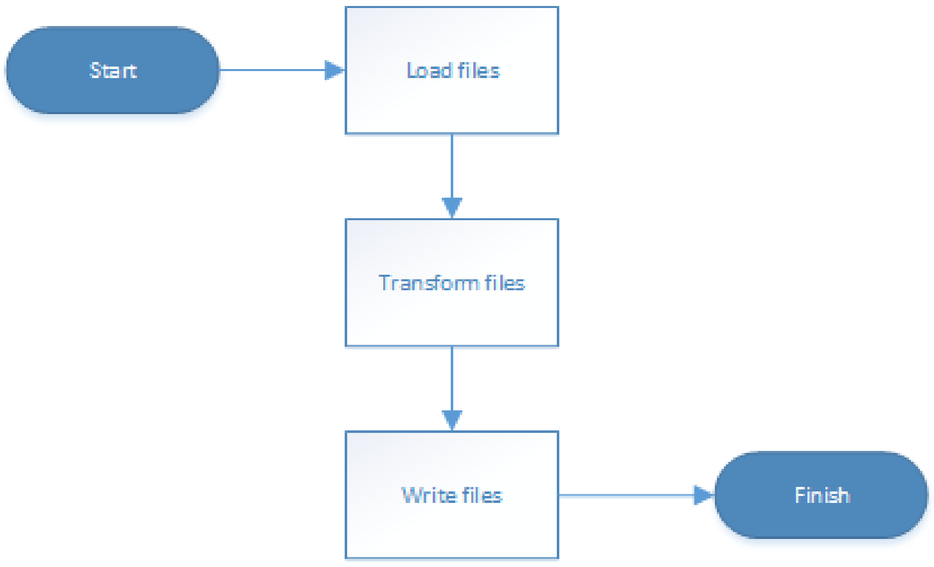


Figure 5.5.: Overview of the necessary processing steps of template files to generate the expected output and write it onto disk.

The way how the template files can be retrieved, how the template files are transformed and how the generated content shall be provided may change. Therefore the IFS should provide extension points and interfaces which allow a customization of these three steps.

For the processing of a template file three informations are necessary First it needs to know how the templates can be loaded and where they are located. In template files predefined place holders are replaced by content mapped to those place holders. So next it needs to know how the templates shall be processed and which place holders shall be replaced by which content. Finally it needs to know where the generated content shall be stored. The location to retrieve or store files can be as simple as a directory on the current file system, but it can also be a zip file or a place on another server.

It is also important for the IFS to provide an API for flexible selection of template files and their target destination. This is a major issue with the Maven archetype which does not allow to generate files at custom destination (**??**). A good API should support several use cases for processing files for code generation. The API should support the selection of either single files or of multiple files at once. Multiple files could be selected by selecting a single directory and specifying which files shall be either included or excluded. Additionally the API should provide the capability to either transform the template files based on predefined place holders or to just copy them. To be able to write file the API has to provide the possibility to specify the target location. If a just a single file was selected for generation an shall be renamed a new file name has to be given. If multiple files were selected and shall be renamed the user may want specify a regular expression which could be used to rename a complete file name or just parts of it.

When generated files are written from the IFS to their original place, conflicts with existing files may occur. Especially if files are written into an existing development environment this can happen. Because the incremental usage of the generator application is a main requirement this is an important use case which has to be handled well. There are two possibilities which have to be considered. If files are generated into a new development environment, then a new directory shall be created and no conflicts can happen. If file are generated into an existing development environment, it is expected to be managed by a version control system (VCS), e.g. by Git Common VCS have the possibility to show the difference between committed files and modifications applied to them. So it is not necessary for the code generator to give this information in case generated file would overwrite existing ones. But to give the user control over file conflicts and increase the trust in the generator that nothing unpredicted may happen, conflicts has to be shown to the user. The user shall decide if the existing or the generated file shall be taken or even both by renaming the generated file.

5.3.5. Generation Domain Specific Language

Up until now separate APIs necessary for different task of code generation were presented. Developing a common language for code generation will make it easier for different stakeholder to talk about concerns of code generation. A domain specific language (DSL), which is using these terms to wrap the different APIs and providing a consistent way to access those APIs can ease the task of writing code generators. The DSL has to integrate APIs from the generator framework but also from the architecture and technology layer. Different generators will need different architecture pattern and technology APIs. So it should be possible for the generator developer to extend the DSL based on required architecture and technology during the development of a code generator. A DSL can be extended statically or dynamically. It depends on the preferences of the generator developers and the capabilities of the chosen programming language which way is more favourable.

5.4. Architecture and Technology Layer

The second layer from bottom is the application and technology layer. An architecture structures a software system components. If an architecture pattern shall be used for the architecture of multiple software systems, it can make it easier for software developers if a common naming strategy of the architecture components and a common codeline organization exists. Additionally the naming strategy and codeline organization can be extended to other projects like infrastructure projects, which are necessary to test and run the software systems. So the development of those projects can become easier for operationals too and be consistent in naming compared to the software projects. Using a common naming strategy and codeline organization of those within the development environment makes it easier for software developers and operationals to work with the generated code assuming that the naming strategy codeline and organization is already known.

The purpose of the architecture and technology layer is to provide an API for the generator developer which supports the development of generator according to naming strategies and codeline organization conventions. The API for software architectures needs a way to select the components of an architecture pattern or projects including implementations of technologies which are necessary to run a software system. Based on the selected component or technology necessary information like the project name, the name space and the project location in the development environment have to be provided. Based on the chosen component or project the amount of information, which can be provided, may differ. It depends also on used technologies, especially build tools which are strongly coupled to a development environment. The API for technologies needs to provide functions to alter the generated content which is influenced by the generation of architecture components and projects in the development environment. It may happen that changes in the development environment may require the adaptation of files in existing projects. This may be true especially for build related files and projects. If a new project is added it has to be added into the build process. Therefore an API for the build technology is necessary providing a function which allows to add new projects in case one is generated. Otherwise the software developer has to do it manually.

5.5. Generator Layer

On top of the application and technology layer is the generator layer. Every generator will be part of this layer. A generator is the piece of code implementing the generator life cycle provided by the generator framework. The whole application is thereby a composition of several generators. A single generator consists of template files and code performing the generation process.

5.5.1. Separation of the Generator and the Templates

The code generator shall enable knowledge sharing through code generation. The templates contain the implementation knowledge which shall be shared. So one of the two core concepts of this layer is the separation of generator and templates. A generator is coupled to the structure of the template files. But not every modification of a template file may require adaptations of the generator code if the new version of the template files can be picked automatically by the generator. So the generator and templates have a different development life cycle. Templates shall be stored in an own project to ease their development and to make them independently accessible from the generator code. This means that both have their own version and a unique identifier. It is up to the implementation if the same or different mechanism are used to release and access the generator and the template files.

Additionally software developers shall be able to make own changes to the templates and use them with the generator. So its not just one version of the template project which will be developed but each developer can have and use its own. In most cases software developers should use a shared version of the template project, but to try out custom modifications a copy of a certain version of the template project has to be provided to the software developer either by an own identifier or a custom version of a template project.

5.5.2. Software Project as Template

Templates are very similar to normal source code. In most cases they contain predefined place holders which can be used to insert custom content. These place holders have a custom syntax resulting in the fact that the template can not be compiled any more. To ease the development of the template project it shall be a normal software projects. The information about the changeable parts shall in a template file shall be stored in a separate file. Because templates are normal software projects now the can be treated the same way. This includes the compilation of a template project but also IDE support for its development. Additionally a template project can be managed by a VCS.

If a generator would directly reference the text strings in an template file both would be strongly coupled. As a layer in between a mapping of text strings in a template file shall be mapped to keywords. The generator shall reference only the keywords and not the text strings directly. This mapping has to be stored in the template project itself in a template description file at a location which is known by every generator by convention. Optionally the description could contain a default questions which could be used by the generator. Additionally the description could contain information about the scope of the place holder so that template files can be included or excluded. It could be useful to either specify just a file or directories or files. If a code generator implementation support this concept the template developer can make more changes to the template without breaking the template processing.

5.6. Generator Application Layer

The last layer is the generator application layer which contains the generator applications. A generator application has two responsibilities First it has to bundle generators into one generator application, which are useful to a group of software developers, and provide a way to execute them. This selection of useful generators has to be done by the software architect. The second responsibility is to the wrap multiple version of the selected generators. If a generator is executed by the generator user without further information then the newest generators and the newest templates shall be used. But the generator user shall be able to specify a generator and also a template project The generator application has to pick the right generator version. Since getting the template project is the responsibility of the generator, the template version has to be passed to the the generator, when it is executed by the generator application. Additionally every generator application needs a way to pass arguments for user queries to the generators.

6. Realization

I don't know
if it's what you want,
but it's what you get. :-)

Larry Wall

Contents

In this chapter an implementation of the architecture and concepts for the code generator from chapter 5 called Pagen is presented, which is necessary for their evaluation. Pagen is intended for software development teams at the KISTERS AG who have to implement software system following the PAP.

Three technologies used at KISTERS AG influenced the implementation. Java is used as programming languages for PAP software system. Maven is the build tool used for all projects. And Git is used as VCS for those projects with Gitlab as a web front end to access and manage those repositories.

6.1. Implementation of the Generator Framework

The generator frameworks implements the generator life clycle, the IFS and the user query API.

The generator life cycle is specified as a Java interface and implemented as an abstract class for generator developers to reuse. The Java interface is shown in listing 6.1. Task is another interface which allows the execution of generators by a generator application. The InputMap is used to store the answer of user queries and to pass those between generators. The TemplateProperties class is used to store the template model. Every generator has its own TemplateProperties instance.

```java
public interface Generator extends Task {

    TemplateProperties configuring(InputMap initialInputMap);

    void writing(Path destinationPath);

    void finalizing();
    }
```

Source Code 6.1: Generator.java

The detailed workflow including the user interactions is shown in figure 6.1. The workflow is separated into actions done by the generator framework, a generator and a generator user. The three hook method representing the three phases of the generation life cycle give control from the generator framework to the specific generator. The user only has to select a generator, provide necessary input and resolve file conflicts. Composable generators have to execute the hook methods of its sub-generators manually at the moment.

The user query API shall help generator developer to prompt the generator user for information. An Java interface for the user prompts is specified and shown in listing 6.2. Every prompt has a method to return its message or question, which will be displayed to the generator user, a method to return the keyword, which will be used to store the user input, and a method executing the prompt and returning the user value. Five different implementations are provided including the four suggested ones from section 5.3.3. Additionally a fake user prompt was added which can be used by the generator developer to add arbitrary values into the resulting map is added. The Prompt Java interface from listing 6.2 is not intended to be used directly by the generator developer Instead a builder pattern is applied on top of this interface and its implementation [**folwer2010**]. For the user query API the builder pattern is used as Java class UserQueryBuilder. For every prompt two method are added to the UserQueryBuilder. A static method for starting a user query and a member method of the UserQueryBuilder, which can be used to chain prompts. Additionally the builder class has a merge function taking another UserQueryBuilder as a parameter. The merge function takes the prompts from the added UserQueryBuilder and adds them to the current one. Finally the UserQueryBuilder has two methods to execute the user queries. The first one takes no parameters and just returns an new InputMap containing the user input. This is a key-value store of the key of prompts and the value added by the generator user. The second execute method from the UserQueryBuilder needs an InputMap as parameter. In both cases prompts are skipped if their key already exists in the InputMap. Additionally there is helper class which can be used to parse the input of the generator user given as argument when executing the generator application. It has just one parse method taking an array of string and returning an Input map.

```java
public interface Prompt<T> {

    String getName();

    String getMessage();

    T prompt();

    }
```

Source Code 6.2: Prompt.java

The implementation of the IFS is based on the concept from section 5.3.4 and adapted to the needs of a code generator for the PAP at KISTERS AG. The template projects will be stored in Git. IFS provides a template loader to access Git repositories via Gitlab as Zip files and stores them in a temporary directory on disk. Additionally it has two ways two template files are processed. The first approach replaces strings in a template file directly by using the content stored in the template model. Processed template files are stored in memory. The original template files in the temporary directory are not altered and can be reused with other template models. A second approach first transforms the template files into Apache Velocity template files and uses its template engine to generate the final content. One possible for Apache Velocity to access the template files is to store them on disk [Apa]. At the moment the destination of a code generation can only be the disk of the generator user execution the code generator. But it is possible to provide other implementations. For selecting template files and processing them a fluent API is provided for the generator, which allows to chose a destination and target folder and if the template files shall be just copied or transformed [Fow10].

6.2. Implementation of the Architecture and Technology Layer

At the moment two APIs are provided, one for the PAP and one for creating AngularJS components following a naming convention which is used for the web component of the PAP. Because the naming strategy and codeline organization for AngularJS web application is only used for PAP software system, both API are placed in a single project at the moment. The implementation of the PAP API supports the generator user with four different Java classes. First it provides an AbstractPortsAndAdaptersGenerator which is an extension of the AbstractGenerator from the generator framework adapted to the the PAP. The AbstractPortsAndAdaptersGenerator contains a user query sequence with question, which has to be answered by any generator, who wants to generate PAP components or related projects. Additionally the Gitlab name space and repository name of the default PAP template project is already available. Next the generator developer is supported by classes with factory methods for the components of the PAP and infrastructure projects which can be used to build, test and deploy a software system. Factory methods are a design to abstract the creation of objects [Gam+95]. These factory methods return NamingConvention objects which is the third Java class providing supported. It and contains the name and and the path of directories according to the naming strategy and codeline organization convention. The last class supporting the generator developer is a helper class which can convert a NamingConvention object into a TemplateProperties object, which will be used to transform template files. The helper class knows about the place holders in the Template Project which are important to any project following PAP and makes use of the user input which is prompted by the user query provided by the AbstractPortsAndAdaptersGenerator. It will use the NamingConvention to map the user input to place holders.

6.3. Implementation of the Generator Layer

First of all generators for all the projects available via the PAP Maven archetype were created. During evaluation of the generator framework and the PAP API other components and projects were added. Figure 6.2 shows an overview of the available PAP components in Pagen. It extends the functionalitly of the PAP Maven archtype by provding another secondary adapter using the Java Persistence API is a standardized API to access and store data from a SQL database [Bau+15].

Figure 6.3 shows the complete overview of the current PAP software system projects with all dependencies. Except the Arc42 documentation and the acceptance test project all of them are available. Especially system deployment projects was extended which were not supported by PAP Maven archetype. An Omnibus Installer project were added, which can be used to create windows installers, and a Chef recipe project, which can be used to provision a software system necessary to run the application.

The KISTERS AG uses Git as version control system and Gitlab to access and manage Git repositories via a web front end. Gitlab provides a very simple way to download whole repositories as single zip files. therefore Git and Gitlab was chosen to store the template project. At the moment it is one single repository containing all the template files. Repositories in Gitlab can be accessed a name space and a repository name. It possible to manage template releases by tags, to create new branches, and to fork whole repositories. Especially the forking of the template repository is very convenient way for software developers to modify an own version od the template project. In fact it is hoped that this capabilities will increase the collaboration of developers for template projects.

6.4. Implementation of the Generator Application Layer

For the evaluation of the code generator architecture and concepts only one generator application was developed. The generator application has to wrap the selected generators and executed the selected version by the generator user. Instead of implementing an own wrapper the generators will be executed as a Maven plug-in. The plug-in is called Pagen as a short hand for ports and adapters generator. Maven can be expected to be installed on all software developer computers. To execute the generator application Maven is executed with the name of the plug-in as first argument. A released version of the plug-in is bound to a fixed generator versions. So to select a particular generator version, the matching plug-in version has to be selected. The generator application developer has to decide, which generator are to be used with a new release of the plug-in. If the newest released version of the generator plug-in shall be used, no version has to be specified when the Pagen plug-in is called. If a specific version of the plug-in shall be used, it has to added to the execution command of the plug-in. Maven will handle downloading the right generator version and its transitive dependencies including the generator framework. So the whole task of managing the right generator version is handled by Maven.

If no further arguments are provided when the Pagen plug-in is executed, the user query API is used to display a list of all generators. The generator user can select one of those and the selected generator is executed. According to the concept of a generator application it shall be possible to select a template version. Templates are stored in a Git repository and can be accessed via a Gitlab server. The server URL, the name space of the template repository, its name and a reference are possible arguments.

It is possible to select the generator not only interactively but to give the short name of a generator as argument. The short names of generators are displayed in square brackets in the list of generators if the plug-in is executed without specify a generator as argument. Additionally it is possible to give the whole information which a generator would ask the user by the user query API as arguments. Therefore it is possible to execute a generator without the need of user interaction.

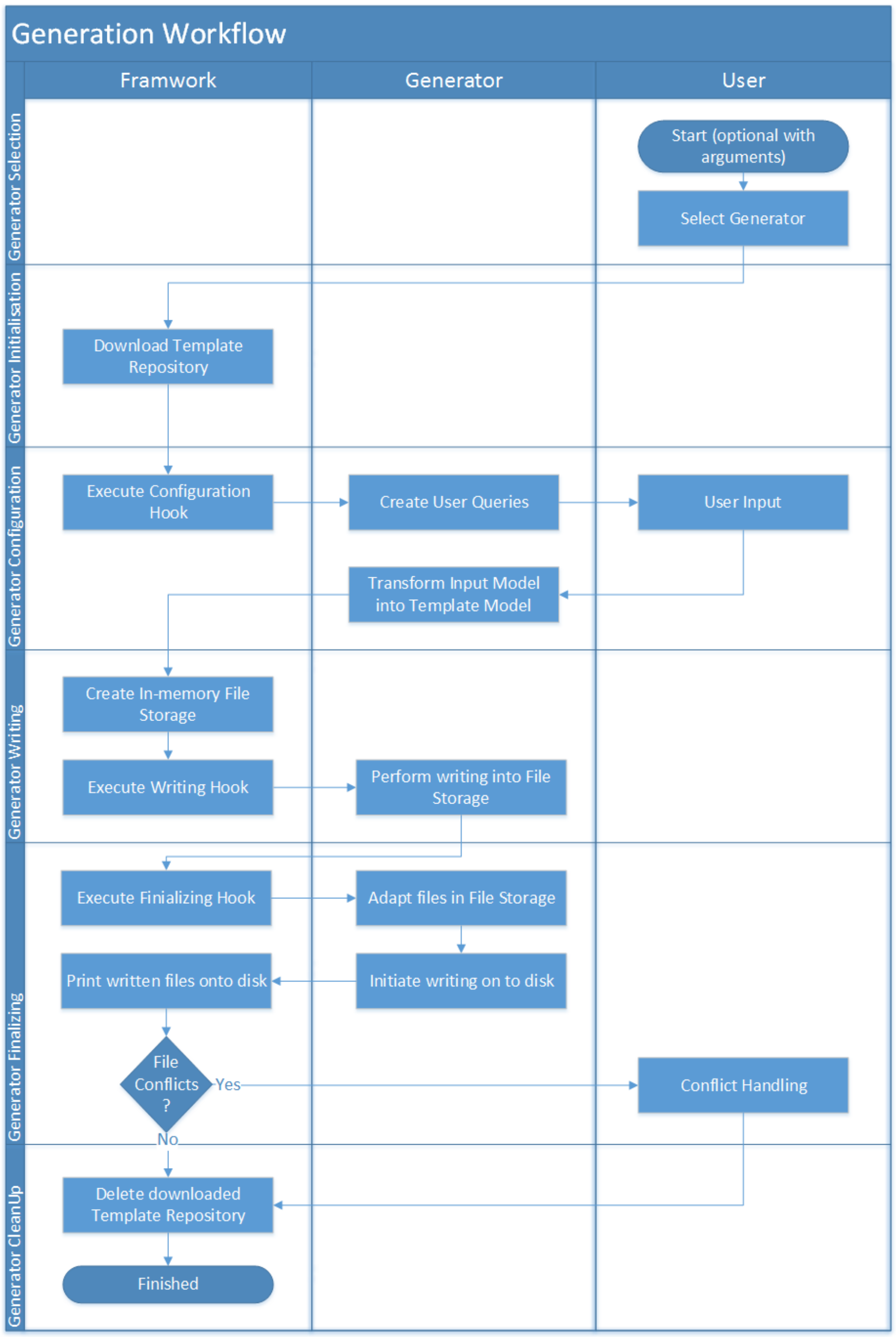

Generation Workflow
Framwork
Generator
User
Generator Selection
Generator Initialisation
Generator Configuration
Generator Writing
Generator Finalizing
Generator CleanUp
Start (optional with arguments)
Select Generator
Download Template Repository
Execute Configuration Hook
Create User Queries
User Input
Transform Input Model into Template Model
Create In-memory File Storage
Execute Writing Hook
Perform writing into File Storage
Execute Finializing Hook
Adapt files in File Storage
Print written files onto disk
Initiate writing on to disk
File Conflicts ?
Yes
No
Conflict Handling
Delete downloaded Template Repository
Finished

Figure 6.1.: Detailed Generation Workflow

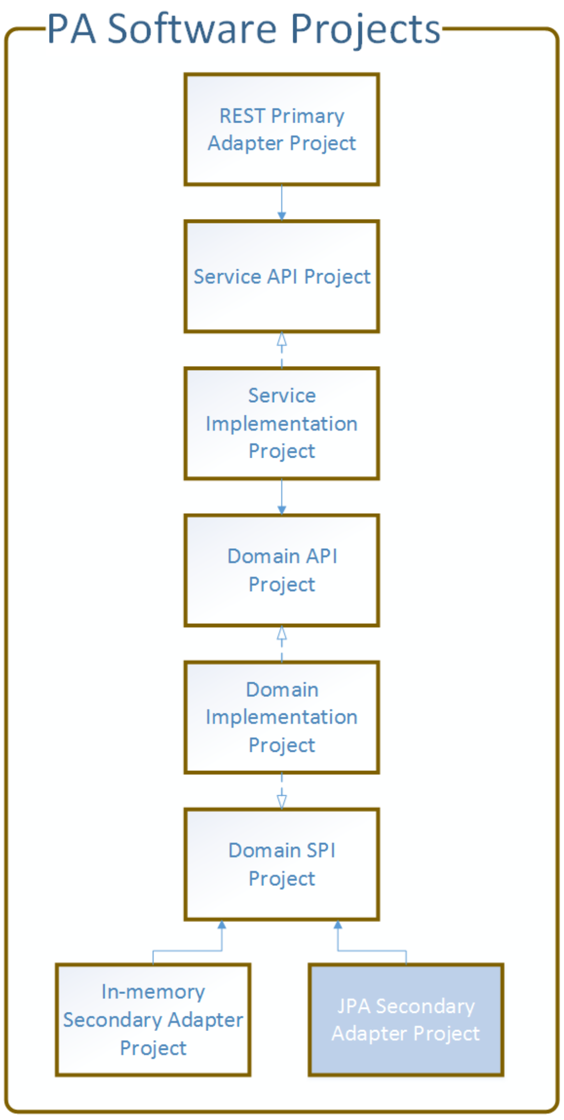

PA Software Projects
REST Primary Adapter Project
Service API Project
Service Implementation Project
Domain API Project
Domain Implementation Project
Domain SPI Project
In-memory Secondary Adapter Project
JPA Secondary Adapter Project

Figure 6.2.: The model of the software projects in Pagen.

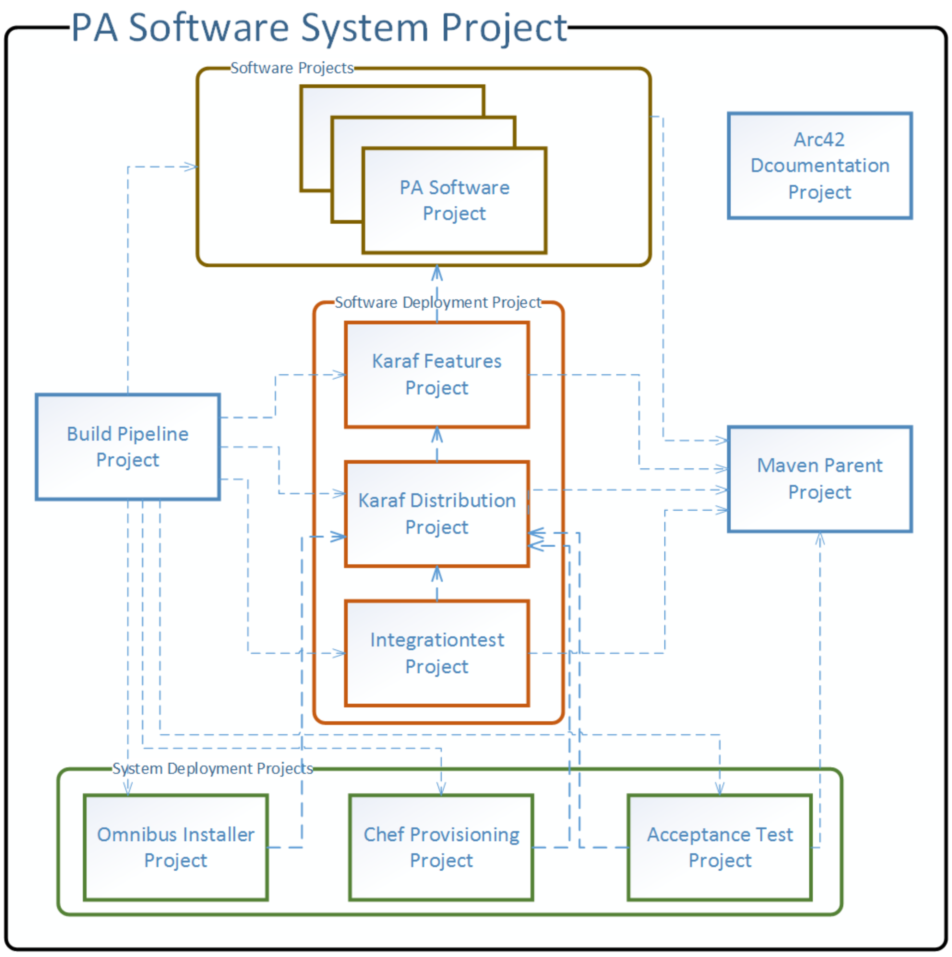

PA Software System Project
Software Projects
PA Software Project
Arc42 Dcoumentation Project
Software Deployment Project
Karaf Features Project
Build Pipeline Project
Karaf Distribution Project
Maven Parent Project
Integrationtest Project
System Deployment Projects
Omnibus Installer Project
Chef Provisioning Project
Acceptance Test Project

Figure 6.3.: The model of the software system in Pagen.

7. Evaluation

It's not a bug - it's an
undocumented feature.

Author Unknown

Contents

In this chapter the architecture and the concepts of the code generator are evaluated. First the implementation of the code generator described in chapter 6 is evaluated against the requirements from section 2.5. Afterwards as a part of the case study at KISTERS AG qualitative surveys with stakeholders are presented which have reviewed, used and extended the generator. Finally the benefits and limitations of the code generator are discussed.

7.1. Evaluation of the Implementation Regarding the Requirements

The requirements from section 2.5 are separated into four groups. Each requirement of those categories are checked against the implementation. the same order of the categories is chosen as in section 2.5.

7.2. Architecture support

Software architecture shall be supported by code generation of an exemplary application implementing the architecture. This requires the architecture to be a pattern which can be reused for multiple software system.

For the PAP and the architecture of an AngularJS web application the code generator implementation provides support for naming strategies and codeline organization. Additionally new ones can be added at any time. So requirement A.1 is implemented.

Code is generated for three different viewpoints, which are related to different stakeholders and different concerns. So the generated code serves different purpose like implementing functionality or automating the provisioning of servers. Furthermore, there are no limitations for any technology and templates can use any programming language or tool. Within the templates the four different programming languages Java, Groovy, JavaScript, and Ruby are used and three different build tools. This makes the code generation very heterogeneous and fulfils requirement K.2 and K.3.

Developing a new generator is very easy due to the provided generator framework APIs, the generation life cycle and the architecture support. New code generators can be used to implement new architecture component or new technologies, which fulfils requirement A.4 and A.6. Requirement A.5 requests a simple way to extend the generator by new architecture patterns. The implemented architecture can be used as guidance. They illustrate, which classes have to be implemented for a good support of an architecture pattern. This should be sufficient for now but maybe improved later on when more architecture patterns are added and a better understanding for this kind of support can be derived.

Maven will automatically download the specified version of the Pagen Maven plug-in. It is even not necessary to install it in the first place because it is all handled by Maven. Only Maven has to be available on the computer. So requirement A.7 is implemented by Maven. If no version is specific the newest one will be picked. However it is not possible to pick a generator version freely, just the ones provided as Maven plug-in. Still this should be sufficient to fulfil requirement A.8. Being able to specify any generator requires that the generator versions are well documented. Otherwise no well-informed choice of a generator version would be possible. Also picking the generator version, which work well together, for an incremental usage may become very hard. Having a selection of generator bundled into a Maven plug-in will increase the reliability of the code generation. Documentation is still necessary but should be easier because it is less distributed.

The AbstractGenerator class provides a code generation life cycle with clear responsibilities of the life cycle phases implementing requirement A.10 The AbstractPortsAndAdaptersGenerator also provided convention for PAP specific user query. So conventions for code generation, which are requested by requirement A.9, are implemented by the generator framework and the architecture and technology layer.

The IFS implements requirement a.11 and is therefore very flexible regarding the location of the generated content, which can be determined dynamically during execution of the code generator. Additionally the IFS can be extended to not only write onto disk but also create Zip files with the generated content.

The requirements A.12, A.13, A.14 and A.15 are about PAP support. All of them are implemented by the code generator. It is possible to generate exact the same code as provided by the Maven archetype and even more. Every PAP component can be selected solely for code generation. This is implemented by breaking the generator into multiple small ones and providing the possibility to compose generators for e.g. generating the whole application. Additionally the generation of subslices is possible resulting in full support of the PAP naming strategy and codeline organization. A new secondary adapter using the Java Persistence API, a project which can be used to develop Chef recipes and an Omnibus project which can be used to create installers were added to the generator. But there are still some other suggestion, which were not implemented yet and are future work. But the three new projects were the most important ones at the time of development.

7.3. Incremental Code Generation

Tools also need to be able to integrate in to work process of the users without enforcing unwanted changes of this process. By making the code generator incrementally usable the generator can be used at any time. To achieve this six requirements regarding incremental code generation were collected and are shown in table 2.2.

The code generator can be used to generate the PA components either into new projects or into existing projects. Additionally new infrastructure projects provided by the code generator can be generated at any time into an existing development environment. Therefore requirement I.1 and I.2 are fulfilled, but the implementation has some limitations. If a new project is generated then the Maven build file expects another project called parent containing common configuration and dependencies for the build process at the same directory level. Additionally any new project has to be added to a super seeding Maven build file and the build pipeline manually. If a new PA component is generated which implements a provided service or provides one, it has to be added manually to the OSGi configuration file, which is used for modularization at KISTERS AG. So at the moment manual work is still necessary which can be improved in future versions of the code generator.

The generator user can decide if the exemplary code shall be generated or not. If the user decides not to only the project following the PAP naming strategy is generated. So the code generator also fulfils requirement I.3. If the generator user decides to not generate a project and no exemplary code nothing at all is generated.

The generator user can either generate single PAP components, a whole slice or something in between with a single execution of the code generator. However, the generator user can not freely choose the needed PAP components and projects and let the code generator generate them with a single execution. To achieve this several executions of the code generator are necessary. Executing the code generator in batch mode can reduce the necessity to add the same content for every execution but still a single generators have to be selected each time. Therefore the code generator does not fulfil the requirement I.4.

To make it easier for the generator developer to provide generators generating more than on project or PAP component, generators are composable as requested by requirement I.5. So to generate a whole slice atomic generators for each of the infrastructure project and the PAP component project are reused. This really makes it easier to maintain generators by reducing the necessity of duplicated code.

The last requirement was about additional support for refactoring PAP components. This exceeds the capabilities of a pure code generator but makes sense as part of architecture support. Software system architecture should be supported by code generation but this support was not limited to code generation. However, the code generator does not implement this requirement. Before this is possible the code generator needs a way to detect PAP components in the current development environment. The PAP naming strategy for projects is not followed completely by a few development environment of existing software system. There was not enough time to get an detailed overview of existing software systems about the naming strategy compliance. Also the naming strategy does not make statements about the fact how projects shall be named if they contain more than one PAP component. Additionally there were not enough information yet if such a feature would be useful to a broad base of software developers because the way how PAP slices are developed is currently changing. Instead of starting with a PAP slice, which has a single project for each PAP component, another development process starts by adding all PAP components into single project and only splits them into multiple projects if they become to big. This shall reduce the overhead of too many software projects. But the code generator reduces the amount of overhead resulting from the creation of new projects. Therefore this feature was postponed as future work. So it has to be decided in the future if the overhead of multiple projects is still too much and the functionality to split projects would be useful.

7.4. Ease of Use

Ease of use is important for code generators, which shall be used for governance and to transfer knowledge [New15]. This was the purpose of the requirements from the category ease of use presented in table 2.3.

The generator user can execute the code generator, which provides a command-line interface, from the console. The whole user interface is textual. User input can either happen interactively or as arguments. Even a combination of both is possible. Everything which is not passed as arguments has to be answered interactively. This provides the most flexibility and it makes it easy to try out different generators with the same input. So requirement U.1, U.2 and U.3 are implemented by the code generator.

Requirement U.4 requested IDE integration via an IDE plugin. Eclipse and IntelliJ Idea are IDEs which are used by software developers at KISTERS AG. Maven is installed on all developer computers and therefore the code generator is available without needing to install it. The effort to get familiar with the plug-in development of two IDEs, then actually developing the plug-ins and maintaining them seemed not worth the benefit. Additionally automatic updates of the generators were not possible requested as requirement A.7. Therefore it would be necessary to implement an own wrapper, which came for free via Maven for the CLI version of the code generator. So requirement U.4 was postponed for now.

Requirement U.5, which is automatic file conflict detection and the possibility to resolve them by the generator user, is implemented by the IFS. This works sufficient but each file conflict has to be resolved by it own. If the generator use could decode once for all file, how file conflicts shall be resolved, it would be a useful extension of the requirement.

Requirement U.6 requests automatic detection of new generators and requirement U.7 request an overview of all generators in a repository. For now both requirements are not implemented. A generator application is a selection of useful generators for a particular architecture. So instead of implementing U.6 and U.7 a different path was chosen, which gives the software architect more control over the available generators which are useful to the software development team. If this is the better approach has to be evaluated over a longer period of time in the future. Therefore at the moment there is no need for requirement U.6, but this can change in the future.

The last requirement from this category U.8 requested Git integration. Around the same time of the requirement interviews an own Maven plug-in was released by a software developer at KISTERS AG for just this purpose. therefore there was no need any more to integrate Git into the code generator.

7.5. Knowledge Sharing

Software developers shall be able to reuse there code as templates to achieve knowledge sharing of implementation best practices. Requirements for this cause were presented in table 2.4.

The template project contains a full exemplary application. Only the new infrastructure Chef project and Omnibus installer project are at the moment mainly generating boilerplate code. The code necessary to create an Chef recipe and an installer for the exemplary application code is still missing and has to be done in future. But for the most part requirement K.1 is fulfilled.

The code of software developers can be directly used as templates without any modification, so requirement K.2 and K.3 are both is implemented too. Text strings in the code which shall be replaced by generator user content have to be defined in a template description file, which was described in section 5.5.2.

Templates are regular source code and they are stored in a Git repository separately from the generator code. Therefore requirement K.4 is fulfilled. The whole generator application including the implementation of generators, which are using the templates, are provided as Maven plug-in. Another way could have been to store the particular generator next to the templates, for example as Groovy scripts. This would make it harder to provide a custom selection of useful generators and ensure the quality of generators. An the other hand this approach would ease the development of new generators. So both ways have their good points. It could be future work to provide a new Groovy DSL for code generation and an implementation which can execute scripts for generator stored in template projects.

At the moment there is only one template repository and so requirement K.5 is not implemented yet. But splitting up the repository into smaller ones would make it necessary to solve some issues with the current code generator first, which would rise up. When the generator is executed the coordinates of a template project, which are the Git name space, the repository name and a reference to a commit, can be passed as arguments. But only one set of those coordinates can be passed as arguments, which is then used for all template repositories. But generators are composable and can make use of other generators. If a composed generator is executed, whose sub-generators have their templates stored in different repositories, there is now way at the moment to pass other template repository coordinates at the moment without breaking the code generation. Software developers could not easily modify the template code any more and use it for generation. As a result the quality of template project regarding best practices is assumed to drop. But hopefully this issue will be fixed in the future an template repositories can be kept smaller and perhaps even reused for more than one generator application.

Requirement K.7 requests that templates can be build, tested and released. For code generation only the source code of templates is necessary. The template project is managed by the VCS tool Git, which persists the state of a template project as commits. It is possible to put references on commits. Therefore being able to put a version as reference to a commit is sufficient as release and fulfills requirement K.6. Of course the templates can be build because they are a exemplary application. But except from the test which are part of the exemplary application there is no further support for testing the generator. Being able to tests part of the generation process itself and not just the generated content could be useful but is still part of future work. Therefore K.7 is not considered to be implemented because testing is an important of software development.

Software developer can fork template repository and use this one instead. A the forked repository will be added to the name space of the software developer and can be referenced by the three coordinates mentioned above. So requirement K.8 is implemented.

At the moment the master reference is used as a default value to get automatic template updates. So requirement K.9 is implemented. However references on other commits still have to be placed or older versions of the template project would not be usable. A better approach would be to use references which refer to a generator version. The generator version could be appended by a template project version. An automatic mechanism could easily pick the highest template version for the particular generator version. If a the template project has to be developed for multiple generator version in parallel because of some breaking changes, Git branches would become necessary. Branches can also be accessed by a reference.

7.6. Case Study at KISTERS AG

In this section the evaluation of the Pagen code generator by employees of the KITERS AG are presented. For the evaluation at KISTERS AG a evaluation sheet was created B.

Evaluation result of a software developer	
Question	Rating
1.1.a	5
1.1.b	-
1.2.a	4
1.2.b	Default input values and remembering of former input
1.3.a	4
1.3.b	Order of generators and adaptation of POM files

Table 7.1.: Overview of the rating for the code generator usage by the software developer.

The questions in the sheet were separated into question regarding the usage of Pagen and regarding its extension. For every topic a rating of implemented features was requested and one question for further comments. Allowed grade for the rating should be between 1 and 5, while 1 meaning a bad implementation and 5 a good one.

7.6.1. Evaluation of the Generator Usage

To evaluate the usage of the code generator a software developer was asked to use the generator and to answer some questions afterwards. First the software developer used Pagen to generated a whole slice first. Afterwards for a new project Pagen was used several times to generate single PAP components. Finally the software developer had to answer question from the the evaluation sheet B regarding the generator. The answer are presented in table 7.2. Pagen got the best rating regarding the ease of installing the generator and good ones for its general and incremental usage. Regarding the general usage, default values and the possibility of the generator to remember former values, were two new requirements, which were requested. Regarding incremental usage a better order and categorisation of available generators were requested and tooling support for Maven. In particular the adaptation of the Maven POM file based on generated content and the existing development environment.

Evaluation of Adding an Infrastructure Generator

To evaluate the development of new generators a operational was asked implement a new generator for a infrastructure project and to answer some questions afterwards. The implementation was done together with the developer of Pagen. The operational had already created a project, which could be used as template project. First these files were added to the template Git repository. Then the content which shall be replaced in the template files was determined and added to the template description file. Afterwards in the project containing all the generators a new generator Java class was created extending the AbstractPortsAndAdaptersGenerator class. A new user query was created for additional and merged with the one provided by its abstract super class. The PAP API was used

Evaluation result of an operational	
Question	Rating
1.1.a	4
1.1.b	IDE plug-ins for the generator
1.2.a	4
1.2.b	Graphical UI
1.3.a	-
1.3.b	-
2.1.a	5
2.1.b	4
2.1.c	5
2.1.d	Recognition of unintended replacement

Table 7.2.: Overview of the rating for the code generator usage by the software developer.

to create a new model and transform the InputMap into TemplateProperties. Then the files, which shall be transformed or copied were selected in the writing method. This was more complex than in the other generators, but the operational was not restricted by the IFS. Finally the new generator was released with a SNAPSHOT version of Pagen to test the new Omnibus project. In a first test it was discovered that more content was replaced than expected because the same text string was also used in other template files. The template project had to be adapted by changing the content and adapting the template description. No new release of the Pagen was necessary. Then the operational was asked to answer the evaluation sheet B. The answer are presented in **??**. Regarding the usage of the generator an IDE plug-in and a graphical UI was requested. Since the incremental usage was not tested, the questions could not be answered. The creation was new generators was evaluated very good. Only a way to recognize unintended replacement of content in template files was requested.

7.6.2. Evaluation of Generator Framework Review

To evaluate the generator framework and the implementation of the architecture and technology layer two reviews were performed with employees at KISTERS AG which are software architects and tooling developers at the same time. They were the only one which had to answer the full evaluation sheet. The review was done together with the Pagen developer. The result is shown in 7.3. The ratings for generator usage nd generator development were similar to the other evaluations. Again a UI and other possibilities to execute Pagen besides Maven were requested. Also default values and the remembering of former input was requested. Then the distribution of template projects into smaller ones, which is in fact requirement K.5, was requested. A completely new request was the one for more documentation of the code generator framework.

Evaluation result of an software architect and tooling developer		
Question	Rating 1	Rating 2
1.1.a	5	5
1.1.b	-	Not only Maven
1.2.a	5	5
1.2.b	Default values	UI
1.3.a	-	4
1.3.b	-	Remembering of former input
2.1.a	4	4
2.1.b	5	5
2.1.c	4	5
2.1.d	Testing and smaller templates projects	
2.2.a	-	5
2.2.b	-	5
2.2.c	5	5
2.2.d	5	5
2.2.e	4	5
2.2.f	5	5
2.2.f	Complex writing phase	More documentation

Table 7.3.: Overview of the rating for the code generator usage by the software developer.

7.7. Discussion

In this section several issues and possibilities for improvement of the code generators are discussed. First the capabilities of the architectural viewpoint regarding code generation are discussed. Then the architecture and technologies layer is discussed and subsequently the impact of changes in the different layers on the other layers. Afterwards the impact of changes in the exemplary application is discussed. Finally benefits and problems of reuse are argued for the code generator.

7.7.1. Code Generation for Viewpoints

The three viewpoints functional, development and deployment have different concerns and stakeholders. Thereby the generated content by a code generator differs for each of these viewpoints. In this section the code generation for the three viewpoints is discussed and its benefit for the generator user.

The functional view is the functional structure based on the requirements of a software system. Requirements are different for each software system. So its is not possible to generated code fitting the functional view of a software system. But it is possible to generate boilerplate code which is necessary for the functional implementation but independent from it. Additionally for the purpose of knowledge sharing it is possible to generate exemplary source code of an hypothetical software system. The purpose of the generated exemplary code is not the functionality itself but the way how it is implemented. It contains best practices and governance as code [New15]. This exemplary code is only useful for software developers, which are new to technology used by the generated code or does not know the governance rules, which their implemented code has be in compliance with. Experienced software developers probably don't want to generate this code every time. If best practices or governance changes there might be the possibility that experienced software developers which don't generate and examine the exemplary code miss the changes. So code generation like it is implemented here can only be a way to experience best practices and governance and is not sufficient to inform software developers about changes. Code generation for functional view might be a way to let software developers see and examine best practices but there are still other mechanism necessary to inform them about those.

The development view is a model of the development environment containing the projects where source code files are generated into. This can be the implementation of the functional view or of the deployment view. Based on the software project model from section 4.2 this can be for example projects containing the exemplary code, projects to test the software system or projects, which implement the models of the deployment view. Depending on the structure of the views and the used technology the number of projects will differ for each system. Additionally software developers, testers and operationals may have different projects just for their own purpose. So based on the development view different projects have to be generated. These generated projects will organize the source code files in compliance to the convention of the codeline model and may contain a build script for automated build, testing and release using the generated source code file structure. In contrast to the functional and deployment view the development view is not directly influenced by the requirements of a software system but by the way how the functional and deployment view are structured. Therefore for a similar functional and deployment view and modularization of source code files the generated projects can be used for the development of a software system itself and are not exemplary projects. This makes the generation of projects useful for beginning and experienced software developers, tester and operationals. There maybe also projects which are intended for the development view itself like a project containing an automated build pipeline or projects managing the dependencies or build tools and plug-ins. These projects may be mandatory for the development environment and for some functional and deployment projects.

The deployment view of a software system contains the hardware nodes and network where the functional elements are deployed at. Software systems will have different requirements regarding the hardware and the network. So regarding code generation this will be similar to the functional view. It is possible to generate boilerplate and exemplary IaC files for projects intended for deployment. The generated deployment code can either use virtual machines on the developer machine or corporate systems provided for this purpose like a database management system (DBMS). Here the same as for the functional viewpoint applies.

So the code generator presented by this thesis is very good at generating the development environment because it is not influenced by the requirements of a software system. Generating boilerplate code for the functional and deployment viewpoint is fine too, but the exemplary application for sharing knowledge and governance will not be generated by experienced software developer or operationals unless they see a reason. Informing them about important changes of best practices and governance by using other channels for communication can be a reason to take a look at the exemplary application again.

7.7.2. Providing Architecture Convention instead of Restrictions

The four layers of the code generator architecture define responsibilities for its modules and allowed dependencies. The separation does increase the development work only imperceptibly, but the benefit from the separated concerns and the possibility to reuse the generator for completely different software system is really great. The generation life cycle enabling composable generators and the APIs make a big difference for easing the generator development. But especially putting architecture and technology APIs in an own layer makes the generator framework more universally applicable without being cluttered by unused architecture patterns and technologies. Software architects, software developers and operationals, which are familiar with the used technologies, and the tooling developers responsible for the generator framework have to work together to improve the APIs in this layer. For a good separation of concern every API of an architecture pattern or technology should be usable independently of the of other APIs in this layer. So this layer will have many but small modules which will be easier to develop and maintain these modules. The approach of supporting the generator developer if the provided conventions are used seems to be the better one then implementing restrictions to prevent the generation of code violating those. This makes it much easier for generator developers to develop code generators if they stick to the provided convention. Restriction limit the functionality. These limitations may effect other areas than were the restriction shall be implemented or there may be reasons for exceptions. Software developers will have to work around the limitations making the development harder. In most cases instead prohibiting one way a preferred one can be made more attractive. Therefore APIs are making the development easier if developers stick to the preferred way. The possibility that violations may occur is higher with this approach and may even not be done purposely every time. Still there may be sometimes a reason for those and having to implement a work around can be frustrating. This shall be prevented.

7.7.3. The Effect of Changes Between the Four Generator Layers

The architecture in section 5.1 separates the modules of a code generator into four different layers. If a module changes, than modules from the same or higher layers have to be adapted if they shall make use of the changes. In this section the effect of changes in the modules from different layers shall be discussed and their influence on other layers.

At the bottom is the generator framework. It has the biggest impact on other modules but itself has no dependencies to any other generator architecture layer. Bug fixes and new features easing the generator development and usage are reasons for changes of the generator framework. If changes in the generator framework are backwards compatible adapting other layers should be easy. Generators may be mostly affected and there may be many generators which have to be adapted. If generators are placed in different modules the necessity to change a generator is only given when it is affected by a bug or want to use a new feature.

On top of the generator framework are the architecture and technology related APIs providing help for the generator developer creating generators which generate code in compliance to the architecture pattern. Every architecture pattern or technology should be in an own module. A change in such a component will affect every generator using this architecture pattern or technology. There may be two kind of changes which have to be handled differently. The first one is the extension of this layer by new components. Since no existing code is modified no generator has to be changed. The second one is the change of architecture conventions. This may require the adaptation of generators using this architecture API and should be avoided because there are projects which may still use the old conventions. Using the new version of the generator may result in generated code which is not compatible with the existing development environment. So two generator version have to be maintained using the different architecture API version until the existing projects using the old conventions are adopted.

On the next level are the generators which are likely to depend on an architecture pattern API. Since all generators belonging to one architecture pattern should depend on the same architecture component, every time such an component is updated it will trigger a new generator development increment. Templates are not inevitably affected by changes but it may be useful to adapt them to new conventions because they represent best practices and governance.

The generator application containing the UI depends on the generators. So it may only need to change if new generators are added or existing ones modified. Before a new generator application is release the quality of the available generators needs to be assured.

So a generator relies on a codeline structure of template. Because of the template description file is doe snot have to rely on template content any more. The separation of generator and template into tw projects can result in the wrong conclusion that they are not coupled any more. Additionally changes at existing code in the architecture and technology layer is problematical. Sticking to the the Open-Closed Principle, which states that classes should not be modified but only extend, can prevent that many generators has to be adapted [Mar03].

7.7.4. Knwoledge Sharing

The capabilities for knowledge sharing seem promising. The forking ability of Gitlab makes it realy easy for software developers and operationals to try out their own modifications. An unresolved issue is the propagation of changes in the exemplary application for the generator stakeholders. These changes are not so severe like the ones discussed above for the development view. The stakeholders have to be informed about those on a different communication channel because it can not be expected that an experienced developer or operational is still generating the exemplary application. A wiki can be used to persist these informations but it will not inform the stakeholders only about important changes. So there may be better ways suited for everyone to inform about code generator changes. A company internal developer newsletter, which is published every few weeks or months depending on the amount of changes and the severity could be more appropriate.

7.7.5. Classification of Generated Code for Software Reuse

Sommerville defines benefits and problems which can arise for software reuse [Som10]. Some of those can also be applied to the code generator and generated code.

Especially three problems can also affect the code generator. Creating and maintaining reusable software can be expensive. This is also true for the code generator. It has to be developed and maintained like any other application. Only if the generator is used by many software developers and operationals the overall development costs may be lower. Providing many and useful generator can increase the number of users. Another problem is the not-invented-here syndrome. The code generator was explicitly designed in a way so that every stakeholder can participate in its development, especially the template development. If software developers think that they can write better templates they are hopefully improving the existing ones and sharing their improvements with others. The last problem applicable to the code generator is the finding the software components which are worthwhile to be provided as generators. By reducing the costs to create new generators and to prepare code for being usable as templates hopefully there are many software components which can be transformed into a generator and pay off its development and maintenance.

But there also benefits resulting from the code generator. First the code generator can make effective use of specialist if they provide their knowledge as generators via an exemplary application adapted to the architecture pattern at use. Then the code generator can accelerate the development. Especially recurring task like creating development environment and boilerplate code of certain projects can be done by the code generator. This are tasks which don't add value to a software project because they implement no direct software system requirement. But still they take time. Standards compliance is another benefit of a code generator, if they can be represented as code. Only using the code generator for this purpose is not enough as already discussed in section 7.7.1, but it is a good way to experience these standards hands-on.

7.8. Summary

A comparison of the implementation and the requested requirements showed that most of them were implemented. A evaluation of the code generator with software developers, operationals, software architects and tooling developers was favourable for the code generator. Knowledge sharing, implementation governance, architecture support can be provide as generated code. But there are limitations which have to be kept in mind. A good mix of different techniques to achieve those goals can overcome those limitations. Still the possibility that software developers can experience those directly as code on their own computer will improve their adaptation and distribution in a company.

8. Conclusion

If debugging is the process of removing bugs, then programming must be the process of putting them in.

Edsger Dijkstra

Contents

In this chapter the main requirements, concepts and findings from the evaluation are summarized first. Then future work will be represented. This are either requirements, which could not be implemented and evaluated yet, or needs of further evaluation and analisation of the generator usage and the development of new concepts for its improvement.

8.1. Summary

The main issues which resulted in the development of a code generator were the support of creating incrementally a development environment in accordance to an architecture pattern and optional an exemplary application to share best implementation practices and governance. The main stakeholders are the software developer and the operational. To develop a generator, which will be accepted by its stakeholders, ease of use was another big issue. So architecture support, incremental usage, ease of use and knowledge sharing were the four categories of the requirements. In c3 an evaluation of existing generator frameworks came to the conclusion that none of the generator frameworks could be used for the implementation of the code generator, because it would not be possible to implement important requirements. An evaluation of the existing generators gave an overview of the capabilities of code generators. All of those code generators mainly addressed software developers and not operationals. All code generators could be extended so a more heterogeneous could be achieved. But all lacked required capabilities of to enable an easy knowledge sharing. In chapter 4 a more theoretical look on software development environments from a software architect point of view was described. A software project model was introduced to better understand the entities in the development environment, which were defined as projects. Chapter 5 introduced an four layered architecture for a code generator. The most important concepts introduced in this chapter are:

- A architecture independent generator framework
- A generator life cycle
- An API handling the processing of template files
- An own layer for architecture and technology related implementations
- Separation of generator and templates
- Using normal software projects as templates

Chapter 6 introduced a code generator implementation of the formerly presented architecture and concepts. In chapter 7 the implementation was evaluated against the requirements and an evaluation of different stakeholders was presented. The most requirements could be implemented. Only the two important requirements, I.4 and I.2, were not or not sufficiently implemented. Both were also requested by the evaluation with stakeholders by one of them. Additionally a few new requirements came up. Afterwards a broader discussion about the code generator finalized the chapter. Incremental usage supports the required flexibility for the generator usage. The code generator makes it much easier to create development environments, but there are still many ways to improve this. The capabilities for knowledge sharing are implemented and seem promising, but a way needs to be established to inform stakeholders about changes.

8.2. Future Work

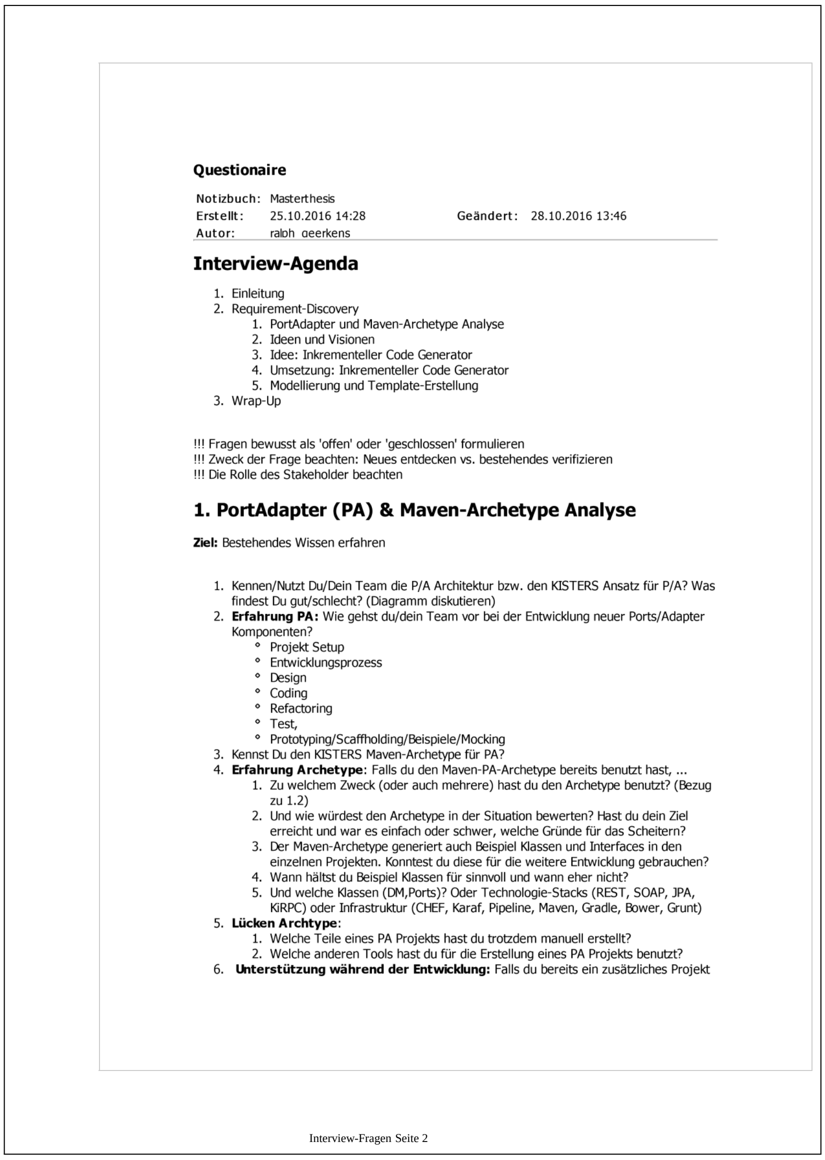
The code generator can be improved in many ways and the most important ones shall be listed here. First of all good documentation for every stakeholder using or developing the generator is required. The documentation should be as easily accessible as the code generator itself. In the generator framework an event-driven concept for propagating the generated projects and existing ones in the development environment would be useful. This could be leveraged by a Maven API in the architecture and technology layer to modify the Maven build files accordingly. If it works well it would reduce the amount of manual adaptation of generated code. The requirement I.4, which requests that it is possible to configure a development environment more flexible could make use of the event-driven project propagation approach.

Then there are a few concepts for the generator layer, which may be useful. First a domain specific language for generator development. A short time during the generator development a DSL for generation was available It made the generator much easier at the time but became unnecessary when software projects could be used as templates. A new DSL could decrease the template processing complexity which was discovered during the implementation of the Omnibus generator Additionally a way to store scripts in the template project and being able to execute them during generation could make the But this feature may require additional security features. And finally distributed template projects would be useful because the template project became quite big and some of the templates are also useful for other team not using the PAP.

Then a new evaluation of the code generator usage and how the acceptance could be improved using the technology acceptance model (TAM) could be performed. A generator which is accepted and used by many user is much more valuable. Other ways of accessing the generator should be evaluated, like a IDE plug-in or web UI for requirement I.4.

All the code generation was limited to development, the functional and the deployment viewpoint. Maybe the code generator can also support other viewpoints like the information layer. So an analysis of the other viewpoints regarding their capabilities for code generation would be interesting too.

A. Requirements Questionnaire



Questionaire

Notizbuch:	Masterthesis		
Erstellt:	25.10.2016 14:28	Geändert:	28.10.2016 13:46
Autor:	ralph geerkens		

Interview-Agenda

1. Einleitung
2. Requirement-Discovery
    1. PortAdapter und Maven-Archetype Analyse
    2. Ideen und Visionen
    3. Idee: Inkrementeller Code Generator
    4. Umsetzung: Inkrementeller Code Generator
    5. Modellierung und Template-Erstellung
3. Wrap-Up

!!! Fragen bewusst als 'offen' oder 'geschlossen' formulieren
!!! Zweck der Frage beachten: Neues entdecken vs. bestehendes verifizieren
!!! Die Rolle des Stakeholder beachten

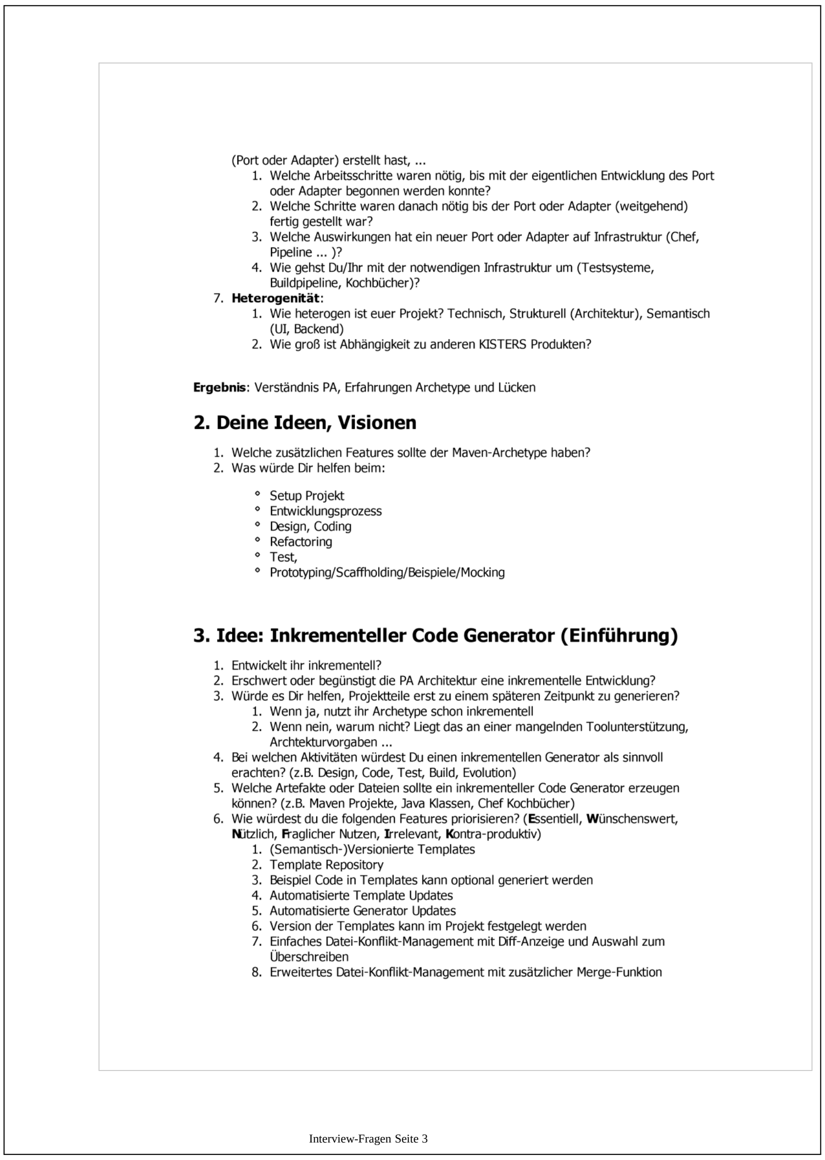
1. PortAdapter (PA) & Maven-Archetype Analyse

Ziel: Bestehendes Wissen erfahren

1. Kennen/Nutzt Du/Dein Team die P/A Architektur bzw. den KISTERS Ansatz für P/A? Was findest Du gut/schlecht? (Diagramm diskutieren)
2. **Erfahrung PA:** Wie gehst du/dein Team vor bei der Entwicklung neuer Ports/Adapter Komponenten?
    - Projekt Setup
    - Entwicklungsprozess
    - Design
    - Coding
    - Refactoring
    - Test,
    - Prototyping/Scaffholding/Beispiele/Mocking
3. Kennst Du den KISTERS Maven-Archetype für PA?
4. **Erfahrung Archetype**: Falls du den Maven-PA-Archetype bereits benutzt hast, ...
    1. Zu welchem Zweck (oder auch mehrere) hast du den Archetype benutzt? (Bezug zu 1.2)
    2. Und wie würdest den Archetype in der Situation bewerten? Hast du dein Ziel erreicht und war es einfach oder schwer, welche Gründe für das Scheitern?
    3. Der Maven-Archetype generiert auch Beispiel Klassen und Interfaces in den einzelnen Projekten. Konntest du diese für die weitere Entwicklung gebrauchen?
    4. Wann hältst du Beispiel Klassen für sinnvoll und wann eher nicht?
    5. Und welche Klassen (DM,Ports)? Oder Technologie-Stacks (REST, SOAP, JPA, KiRPC) oder Infrastruktur (CHEF, Karaf, Pipeline, Maven, Gradle, Bower, Grunt)
5. **Lücken Archtype**:
    1. Welche Teile eines PA Projekts hast du trotzdem manuell erstellt?
    2. Welche anderen Tools hast du für die Erstellung eines PA Projekts benutzt?
6. **Unterstützung während der Entwicklung:** Falls du bereits ein zusätzliches Projekt

Interview-Fragen Seite 2

Figure A.1.: Requirements Questionnaire page 1 of 4



(Port oder Adapter) erstellt hast, ...

1. Welche Arbeitsschritte waren nötig, bis mit der eigentlichen Entwicklung des Port oder Adapter begonnen werden konnte?
2. Welche Schritte waren danach nötig bis der Port oder Adapter (weitgehend) fertig gestellt war?
3. Welche Auswirkungen hat ein neuer Port oder Adapter auf Infrastruktur (Chef, Pipeline ...)?
4. Wie gehst Du/Ihr mit der notwendigen Infrastruktur um (Testsysteme, Buildpipeline, Kochbücher)?

7. **Heterogenität**:
   1. Wie heterogen ist euer Projekt? Technisch, Strukturell (Architektur), Semantisch (UI, Backend)
   2. Wie groß ist Abhängigkeit zu anderen KISTERS Produkten?

Ergebnis: Verständnis PA, Erfahrungen Archetype und Lücken

2. Deine Ideen, Visionen

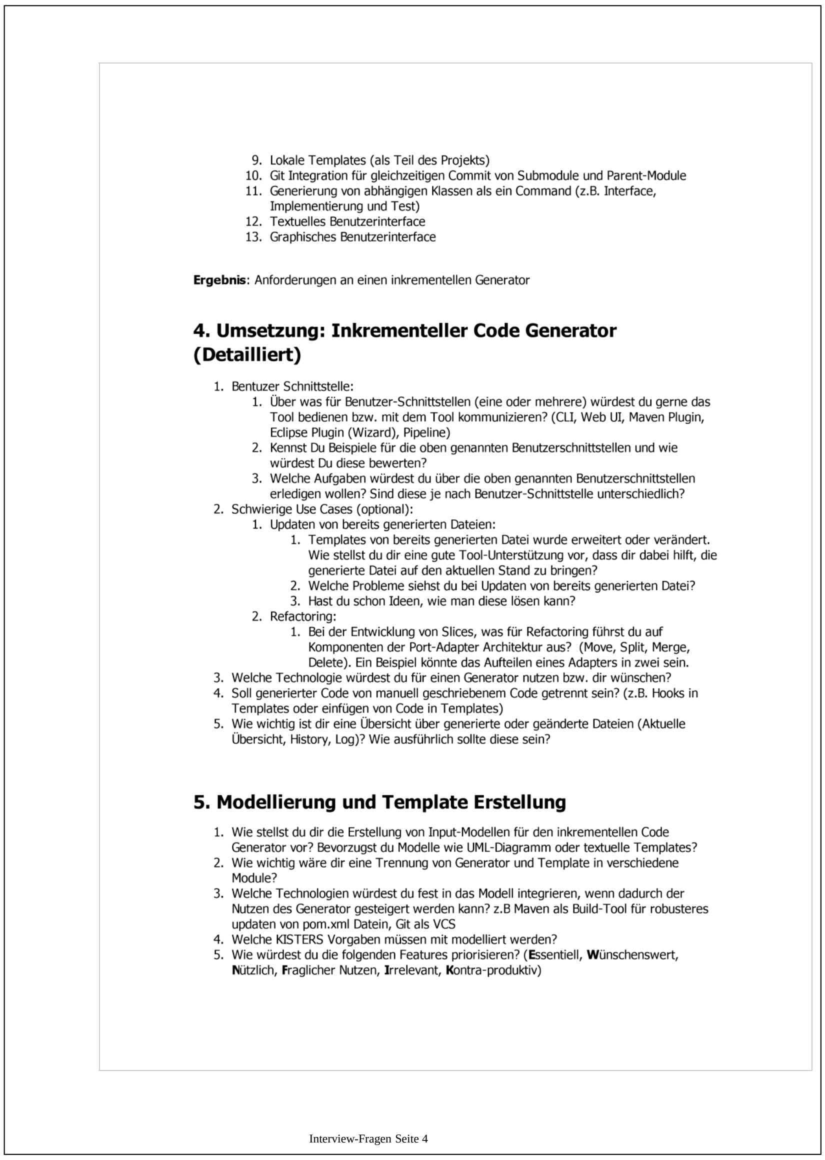
1. Welche zusätzlichen Features sollte der Maven-Archetype haben?
2. Was würde Dir helfen beim:
   - Setup Projekt
   - Entwicklungsprozess
   - Design, Coding
   - Refactoring
   - Test,
   - Prototyping/Scaffholding/Beispiele/Mocking

3. Idee: Inkrementeller Code Generator (Einführung)

1. Entwickelt ihr inkrementell?
2. Erschwert oder begünstigt die PA Architektur eine inkrementelle Entwicklung?
3. Würde es Dir helfen, Projektteile erst zu einem späteren Zeitpunkt zu generieren?
   1. Wenn ja, nutzt ihr Archetype schon inkrementell
   2. Wenn nein, warum nicht? Liegt das an einer mangelnden Toolunterstützung, Archtekturvorgaben ...
4. Bei welchen Aktivitäten würdest Du einen inkrementellen Generator als sinnvoll erachten? (z.B. Design, Code, Test, Build, Evolution)
5. Welche Artefakte oder Dateien sollte ein inkrementeller Code Generator erzeugen können? (z.B. Maven Projekte, Java Klassen, Chef Kochbücher)
6. Wie würdest du die folgenden Features priorisieren? (**E**ssentiell, **W**ünschenswert, **N**ützlich, **F**raglicher Nutzen, **I**rrelevant, **K**ontra-produktiv)
   1. (Semantisch-)Versionierte Templates
   2. Template Repository
   3. Beispiel Code in Templates kann optional generiert werden
   4. Automatisierte Template Updates
   5. Automatisierte Generator Updates
   6. Version der Templates kann im Projekt festgelegt werden
   7. Einfaches Datei-Konflikt-Management mit Diff-Anzeige und Auswahl zum Überschreiben
   8. Erweitertes Datei-Konflikt-Management mit zusätzlicher Merge-Funktion

Interview-Fragen Seite 3

Figure A.2.: Requirements Questionnaire page 2 of 4



9. Lokale Templates (als Teil des Projekts)
10. Git Integration für gleichzeitigen Commit von Submodule und Parent-Module
11. Generierung von abhängigen Klassen als ein Command (z.B. Interface, Implementierung und Test)
12. Textuelles Benutzerinterface
13. Graphisches Benutzerinterface

Ergebnis: Anforderungen an einen inkrementellen Generator

4. Umsetzung: Inkrementeller Code Generator (Detailliert)

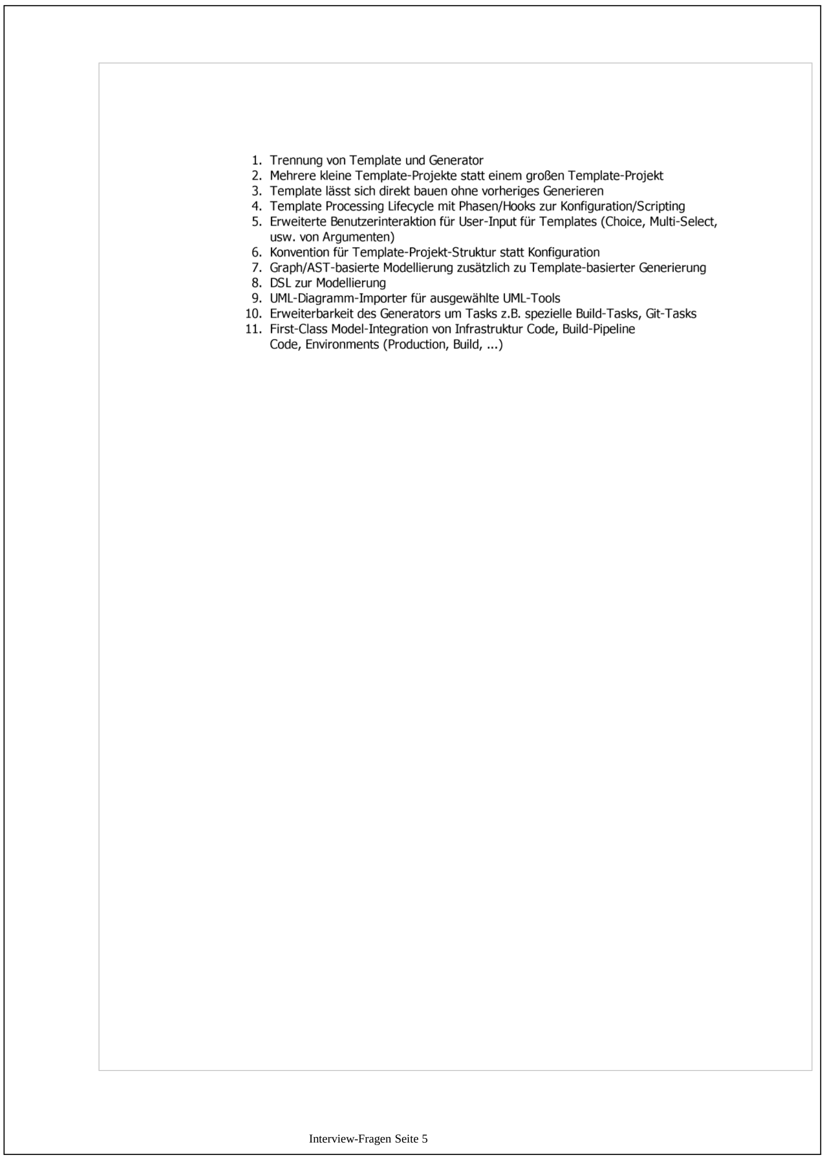
1. Bentuzer Schnittstelle:
    1. Über was für Benutzer-Schnittstellen (eine oder mehrere) würdest du gerne das Tool bedienen bzw. mit dem Tool kommunizieren? (CLI, Web UI, Maven Plugin, Eclipse Plugin (Wizard), Pipeline)
    2. Kennst Du Beispiele für die oben genannten Benutzerschnittstellen und wie würdest Du diese bewerten?
    3. Welche Aufgaben würdest du über die oben genannten Benutzerschnittstellen erledigen wollen? Sind diese je nach Benutzer-Schnittstelle unterschiedlich?
2. Schwierige Use Cases (optional):
    1. Updaten von bereits generierten Dateien:
        1. Templates von bereits generierten Datei wurde erweitert oder verändert. Wie stellst du dir eine gute Tool-Unterstützung vor, dass dir dabei hilft, die generierte Datei auf den aktuellen Stand zu bringen?
        2. Welche Probleme siehst du bei Updaten von bereits generierten Datei?
        3. Hast du schon Ideen, wie man diese lösen kann?
    2. Refactoring:
        1. Bei der Entwicklung von Slices, was für Refactoring führst du auf Komponenten der Port-Adapter Architektur aus? (Move, Split, Merge, Delete). Ein Beispiel könnte das Aufteilen eines Adapters in zwei sein.
3. Welche Technologie würdest du für einen Generator nutzen bzw. dir wünschen?
4. Soll generierter Code von manuell geschriebenem Code getrennt sein? (z.B. Hooks in Templates oder einfügen von Code in Templates)
5. Wie wichtig ist dir eine Übersicht über generierte oder geänderte Dateien (Aktuelle Übersicht, History, Log)? Wie ausführlich sollte diese sein?

5. Modellierung und Template Erstellung

1. Wie stellst du dir die Erstellung von Input-Modellen für den inkrementellen Code Generator vor? Bevorzugst du Modelle wie UML-Diagramm oder textuelle Templates?
2. Wie wichtig wäre dir eine Trennung von Generator und Template in verschiedene Module?
3. Welche Technologien würdest du fest in das Modell integrieren, wenn dadurch der Nutzen des Generator gesteigert werden kann? z.B Maven als Build-Tool für robusteres updaten von pom.xml Datein, Git als VCS
4. Welche KISTERS Vorgaben müssen mit modelliert werden?
5. Wie würdest du die folgenden Features priorisieren? (**E**ssentiell, **W**ünschenswert, **N**ützlich, **F**raglicher Nutzen, **I**rrelevant, **K**ontra-produktiv)

Interview-Fragen Seite 4

Figure A.3.: Requirements Questionnaire page 3 of 4

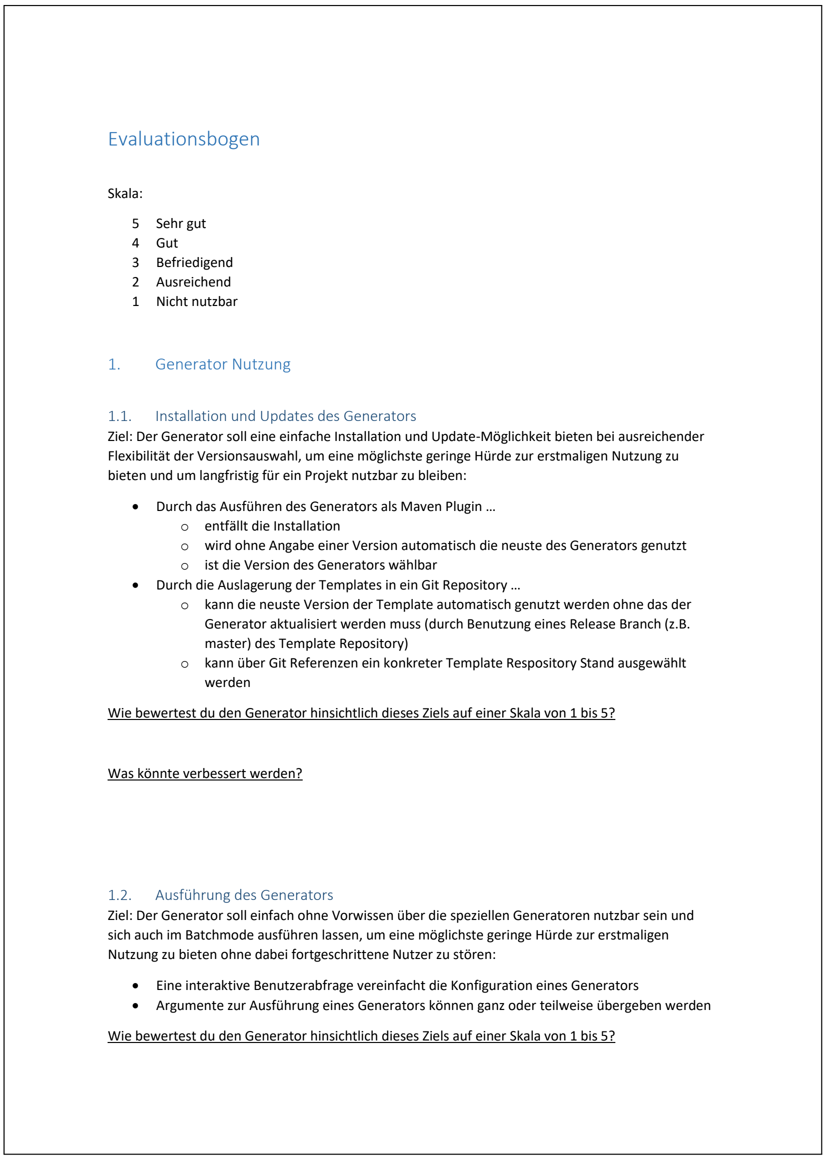


1. Trennung von Template und Generator
2. Mehrere kleine Template-Projekte statt einem großen Template-Projekt
3. Template lässt sich direkt bauen ohne vorheriges Generieren
4. Template Processing Lifecycle mit Phasen/Hooks zur Konfiguration/Scripting
5. Erweiterte Benutzerinteraktion für User-Input für Templates (Choice, Multi-Select, usw. von Argumenten)
6. Konvention für Template-Projekt-Struktur statt Konfiguration
7. Graph/AST-basierte Modellierung zusätzlich zu Template-basierter Generierung
8. DSL zur Modellierung
9. UML-Diagramm-Importer für ausgewählte UML-Tools
10. Erweiterbarkeit des Generators um Tasks z.B. spezielle Build-Tasks, Git-Tasks
11. First-Class Model-Integration von Infrastruktur Code, Build-Pipeline Code, Environments (Production, Build, ...)

Interview-Fragen Seite 5

Figure A.4.: Requirements Questionnaire page 4 of 4

B. Evaluation Sheet



Evaluationsbogen

Skala:

5	Sehr gut
4	Gut
3	Befriedigend
2	Ausreichend
1	Nicht nutzbar

1. Generator Nutzung

1.1. Installation und Updates des Generators

Ziel: Der Generator soll eine einfache Installation und Update-Möglichkeit bieten bei ausreichender Flexibilität der Versionsauswahl, um eine möglichste geringe Hürde zur erstmaligen Nutzung zu bieten und um langfristig für ein Projekt nutzbar zu bleiben:

- Durch das Ausführen des Generators als Maven Plugin …
  - entfällt die Installation
  - wird ohne Angabe einer Version automatisch die neuste des Generators genutzt
  - ist die Version des Generators wählbar
- Durch die Auslagerung der Templates in ein Git Repository …
  - kann die neuste Version der Template automatisch genutzt werden ohne das der Generator aktualisiert werden muss (durch Benutzung eines Release Branch (z.B. master) des Template Repository)
  - kann über Git Referenzen ein konkreter Template Respository Stand ausgewählt werden

Wie bewertest du den Generator hinsichtlich dieses Ziels auf einer Skala von 1 bis 5?

Was könnte verbessert werden?

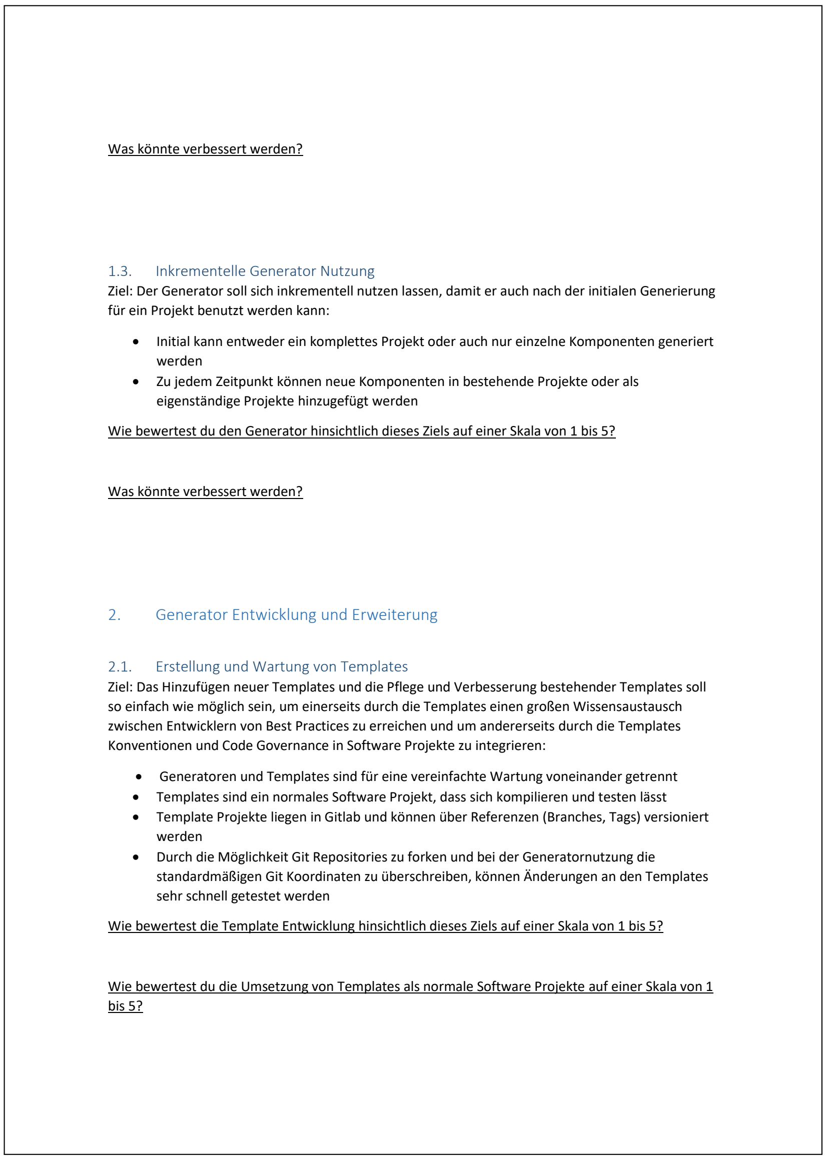
1.2. Ausführung des Generators

Ziel: Der Generator soll einfach ohne Vorwissen über die speziellen Generatoren nutzbar sein und sich auch im Batchmode ausführen lassen, um eine möglichste geringe Hürde zur erstmaligen Nutzung zu bieten ohne dabei fortgeschrittene Nutzer zu stören:

- Eine interaktive Benutzerabfrage vereinfacht die Konfiguration eines Generators
- Argumente zur Ausführung eines Generators können ganz oder teilweise übergeben werden

Wie bewertest du den Generator hinsichtlich dieses Ziels auf einer Skala von 1 bis 5?

Figure B.1.: Evaluation Sheet page 1 of 4



<u>Was könnte verbessert werden?</u>

1.3. Inkrementelle Generator Nutzung

Ziel: Der Generator soll sich inkrementell nutzen lassen, damit er auch nach der initialen Generierung für ein Projekt benutzt werden kann:

- Initial kann entweder ein komplettes Projekt oder auch nur einzelne Komponenten generiert werden
- Zu jedem Zeitpunkt können neue Komponenten in bestehende Projekte oder als eigenständige Projekte hinzugefügt werden

<u>Wie bewertest du den Generator hinsichtlich dieses Ziels auf einer Skala von 1 bis 5?</u>

<u>Was könnte verbessert werden?</u>

2. Generator Entwicklung und Erweiterung

2.1. Erstellung und Wartung von Templates

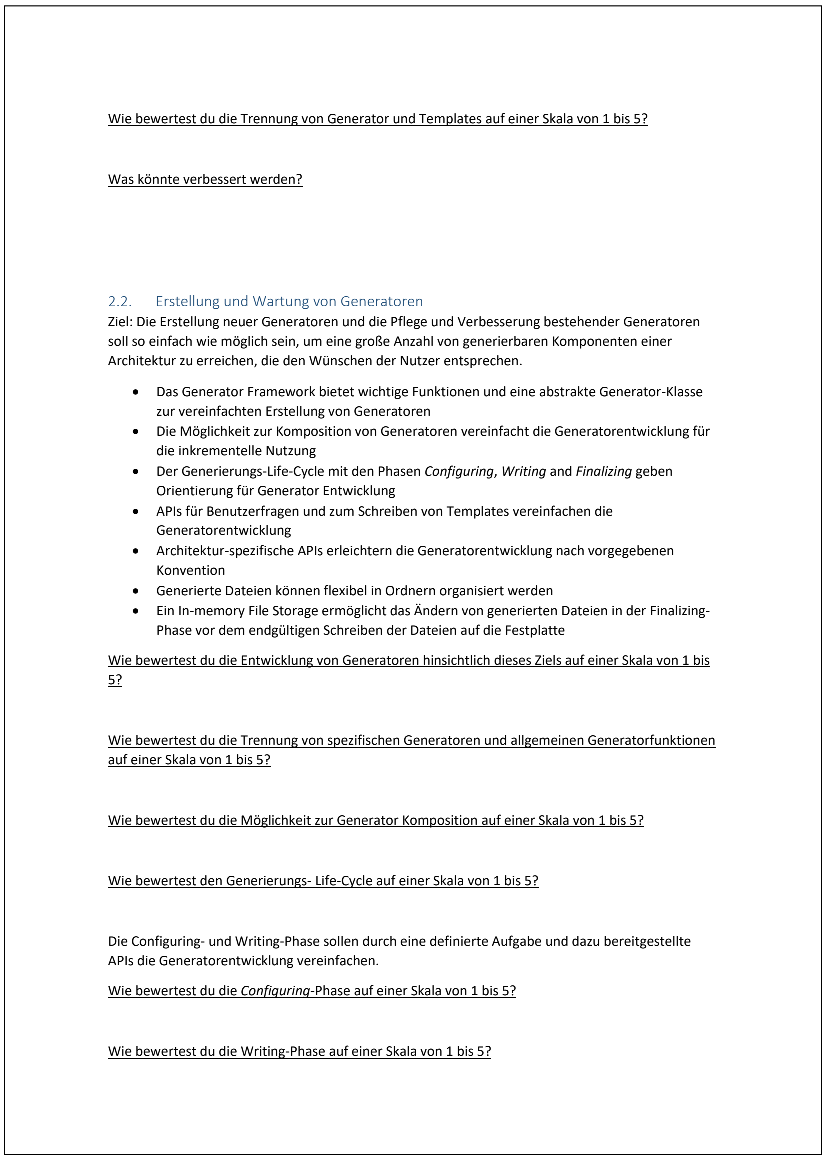
Ziel: Das Hinzufügen neuer Templates und die Pflege und Verbesserung bestehender Templates soll so einfach wie möglich sein, um einerseits durch die Templates einen großen Wissensaustausch zwischen Entwicklern von Best Practices zu erreichen und um andererseits durch die Templates Konventionen und Code Governance in Software Projekte zu integrieren:

- Generatoren und Templates sind für eine vereinfachte Wartung voneinander getrennt
- Templates sind ein normales Software Projekt, dass sich kompilieren und testen lässt
- Template Projekte liegen in Gitlab und können über Referenzen (Branches, Tags) versioniert werden
- Durch die Möglichkeit Git Repositories zu forken und bei der Generatornutzung die standardmäßigen Git Koordinaten zu überschreiben, können Änderungen an den Templates sehr schnell getestet werden

<u>Wie bewertest die Template Entwicklung hinsichtlich dieses Ziels auf einer Skala von 1 bis 5?</u>

<u>Wie bewertest du die Umsetzung von Templates als normale Software Projekte auf einer Skala von 1 bis 5?</u>

Figure B.2.: Evaluation Sheet page 2 of 4



Wie bewertest du die Trennung von Generator und Templates auf einer Skala von 1 bis 5?

Was könnte verbessert werden?

2.2. Erstellung und Wartung von Generatoren

Ziel: Die Erstellung neuer Generatoren und die Pflege und Verbesserung bestehender Generatoren soll so einfach wie möglich sein, um eine große Anzahl von generierbaren Komponenten einer Architektur zu erreichen, die den Wünschen der Nutzer entsprechen.

- Das Generator Framework bietet wichtige Funktionen und eine abstrakte Generator-Klasse zur vereinfachten Erstellung von Generatoren
- Die Möglichkeit zur Komposition von Generatoren vereinfacht die Generatorentwicklung für die inkrementelle Nutzung
- Der Generierungs-Life-Cycle mit den Phasen *Configuring*, *Writing* and *Finalizing* geben Orientierung für Generator Entwicklung
- APIs für Benutzerfragen und zum Schreiben von Templates vereinfachen die Generatorentwicklung
- Architektur-spezifische APIs erleichtern die Generatorentwicklung nach vorgegebenen Konvention
- Generierte Dateien können flexibel in Ordnern organisiert werden
- Ein In-memory File Storage ermöglicht das Ändern von generierten Dateien in der Finalizing-Phase vor dem endgültigen Schreiben der Dateien auf die Festplatte

Wie bewertest du die Entwicklung von Generatoren hinsichtlich dieses Ziels auf einer Skala von 1 bis 5?

Wie bewertest du die Trennung von spezifischen Generatoren und allgemeinen Generatorfunktionen auf einer Skala von 1 bis 5?

Wie bewertest du die Möglichkeit zur Generator Komposition auf einer Skala von 1 bis 5?

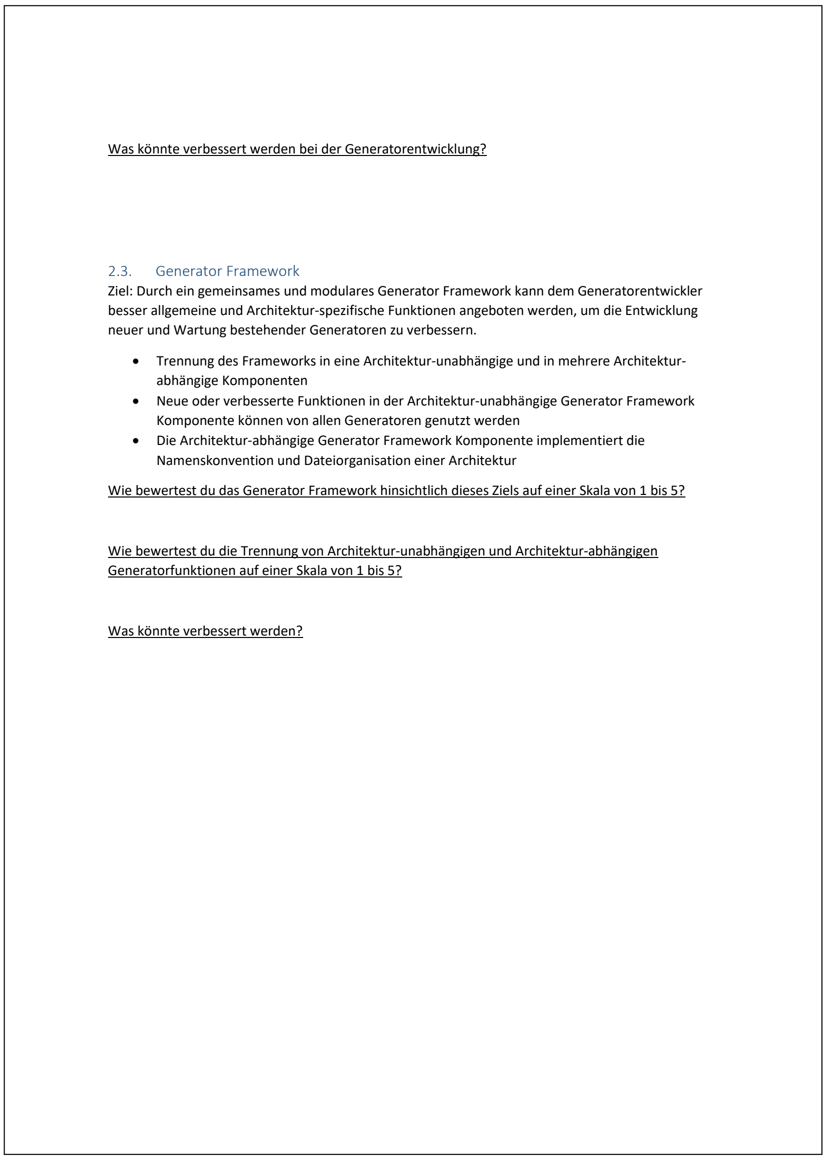
Wie bewertest den Generierungs- Life-Cycle auf einer Skala von 1 bis 5?

Die Configuring- und Writing-Phase sollen durch eine definierte Aufgabe und dazu bereitgestellte APIs die Generatorentwicklung vereinfachen.

Wie bewertest du die *Configuring*-Phase auf einer Skala von 1 bis 5?

Wie bewertest du die Writing-Phase auf einer Skala von 1 bis 5?

Figure B.3.: Evaluation Sheet page 3 of 4



Was könnte verbessert werden bei der Generatorentwicklung?

2.3. Generator Framework

Ziel: Durch ein gemeinsames und modulares Generator Framework kann dem Generatorentwickler besser allgemeine und Architektur-spezifische Funktionen angeboten werden, um die Entwicklung neuer und Wartung bestehender Generatoren zu verbessern.

- Trennung des Frameworks in eine Architektur-unabhängige und in mehrere Architektur-abhängige Komponenten
- Neue oder verbesserte Funktionen in der Architektur-unabhängige Generator Framework Komponente können von allen Generatoren genutzt werden
- Die Architektur-abhängige Generator Framework Komponente implementiert die Namenskonvention und Dateiorganisation einer Architektur

Wie bewertest du das Generator Framework hinsichtlich dieses Ziels auf einer Skala von 1 bis 5?

Wie bewertest du die Trennung von Architektur-unabhängigen und Architektur-abhängigen Generatorfunktionen auf einer Skala von 1 bis 5?

Was könnte verbessert werden?

Figure B.4.: Evaluation Sheet page 4 of 4

Bibliography

[Apa] *Apache Velocity.* `https://velocity.apache.org/` (cited on page 49).

[Bau+15] C. Bauer et al. *Java Persistence with Hibernate.* 2snd. Manning Publications, 2015 (cited on page 50).

[Coc] A. Cockburn. *The Ports And Adapters Architectre Pattern.* `http://alistair.cockburn.us/Hexagonal+architecture` (cited on page 4).

[Col+97] D. Coleman et al. "UML (Panel): The Language of Blueprints for Software?" In: *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications.* OOPSLA '97. Atlanta, Georgia, USA: ACM, 1997, pp. 201–205. ISBN: 0-89791-908-4. DOI: `10.1145/263698.263736`. URL: `http://doi.acm.org/10.1145/263698.263736` (cited on page 26).

[Dca] *The Clean Architecture.* `https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html` (cited on pages 1, 4).

[Dub+08] Y. Dubinsky et al. "UML (Panel): The Language of Blueprints for Software?" In: *Information Technology Governance and Service Management: Frameworks and Adaptations.* IGI Global, 2008. ISBN: 9781605660080 (cited on page 3).

[Ecl] *Eclipse Che.* `https://eclipse.org/che/` (cited on page 30).

[Emb] *Ember.* `http://emberjs.com/` (cited on page 20).

[Eva03] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional, 2003. ISBN: 0321125215 (cited on pages 1, 4).

[Fow02] M. Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley Longman Publishing Co., Inc., 2002 (cited on page 36).

[Fow10] M. Fowler. *Domain Specific Languages.* 1st. Addison-Wesley Professional, 2010. ISBN: 0321712943, 9780321712943 (cited on page 49).

[FS97] M. Fayad and D. C. Schmidt. "Object-oriented Application Frameworks". In: *Commun. ACM* 40.10 (Oct. 1997), pp. 32–38. ISSN: 0001-0782. DOI: `10.1145/262793.262798`. URL: `http://doi.acm.org/10.1145/262793.262798` (cited on pages 36, 40).

[Gam+95] E. Gamma et al. *Design Patterns.* Vol. 47. Addison Wesley Professional Computing Series February. 1995, pp. 1–429 (cited on pages 4, 49).

[Her03] J. Herrington. *Code Generation in Action*. Manning Publications, 2003. ISBN: 9781930110977 (cited on page 13).

[Iso] "ISO/IEC/IEEE Systems and software engineering – Architecture description". In: *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)* (2011), pp. 1–46. DOI: `10.1109/IEEESTD.2011.6129467` (cited on page 25).

[Jhi] *JHipster*. `https://jhipster.github.io/` (cited on page 18).

[Kru95] P. B. Kruchten. "The 4+1 View Model of architecture". In: *IEEE Software* 12.6 (1995), pp. 42–50. ISSN: 0740-7459. DOI: `10.1109/52.469759` (cited on pages 26–28).

[Lig] *Lightbend Activator Documentation*. `https://www.lightbend.com/activator/docs` (cited on page 21).

[Mar03] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003. ISBN: 0135974445 (cited on page 67).

[Mav] *Maven Archetype Plugin*. `https://maven.apache.org/archetype/maven-archetype-plugin/` (cited on page 14).

[Mor15] K. Morris. *Infrastructure as Code*. O'Reilly Media, Inc., 2015 (cited on page 27).

[Muc07] V. Muchandi. *Applying 4+1 View Architecture with UML 2*. Tech. rep. FCG Software Service, 2007 (cited on pages 26–28).

[New15] S. Newman. *Building Microservices*. San Francisco: O'Reilly Media, 2015. ISBN: 978-1-4919-5035-7 (cited on pages 1, 5, 6, 58, 64).

[Rin97] D. C. Rine. "Success Factors for Software Reuse That Are Applicable Across Domains and Businesses". In: *Proceedings of the 1997 ACM Symposium on Applied Computing*. SAC '97. San Jose, California, USA: ACM, 1997, pp. 182–186. ISBN: 0-89791-850-9. DOI: `10.1145/331697.331736`. URL: `http://doi.acm.org/10.1145/331697.331736` (cited on page 1).

[RP12] K. Rimple and S. Penchikala. *Spring Roo in Action*. Manning Publications Co., 2012 (cited on page 17).

[RW11] N. Rozanski and E. Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. 2nd ed. Addison-Wesley Professional, 2011. ISBN: 032171833X, 9780321718334 (cited on pages 2, 25, 28–30).

[Som10] I. Sommerville. *Software Engineering*. 9th. USA: Addison-Wesley Publishing Company, 2010. ISBN: 0137035152, 9780137035151 (cited on pages 1, 67).

[Vli08] H. v. Vliet. *Software Engineering: Principles and Practice*. 3rd. Wiley Publishing, 2008. ISBN: 0470031468 (cited on page 1).

[Wil+16] J. Willis et al. *The DevOps Handbook*. IT Revolution Press, 2016. ISBN: 9781942788003 (cited on page 1).

[Yeo] *Yeoman*. `http://yeoman.io/` (cited on pages 15, 41).

[Zü05] H. Züllighoven. *Object-Oriented Construction Handbook*. Ed. by H. Züllighoven. San Francisco: Morgan Kaufmann, 2005. ISBN: 978-1-55860-687-6 (cited on page 40).