SWC Software Construction

RWTH AACHEN UNIVERSITY

The present work was submitted to the Research Group Software Construction

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

Master Thesis

Continuous Compliance Testing

Continuous Compliance Testing

presented by

Marco Moscher

Aachen, September 18, 2017

Examiner

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Supervisor

Dipl.-Inform. Andreas Steffens

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

Continuous Compliance Testing

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

Continuous Compliance Testing

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, September 18, 2017 (Marco Moscher)

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, September 18, 2017 (Marco Moscher)

Acknowledgment

First and foremost, I would like to thank my supervisor Andreas Steffens for his constant support and valuable feedback throughout the whole period of my thesis. He was always available and offered his knowledge for discussions, explanations and further valuable ideas. All these aspects significantly influenced the proposed approach.

I would also thank Prof. Dr. rer. nat. Horst Lichter for the opportunity to pursue this thesis at the Research Group Software Construction in cooperations with KISTRERS in Aachen. Furthermore, my thanks goes to Prof. Dr. rer. nat. Bernhard Rumpe who has kindly agreed to be the second examiner of this thesis.

Special thanks are in order for Dr. Heinz-Josef Schlebusch for his cooperation and for Oliver Farr for spending the time introducing me to *ProCoS* and providing all requested material to conduct the case study. Additionally I would like to thank all colleagues from KISTERS for their support during the complete thesis.

I thank Jan Pennekamp for proofreading this work. Last but not least, I want thank to my parents, family and especially Marie-Louis Mersch for their continuous support during my whole studies.

Marco Moscher

Abstract

A correct system configuration is a crucial aspect to yield a secure *Environment* because insufficient or even wrong system configuration can impact the overall systems and application's security as reported in the *Top 10 Application Security Risks* paper by the *OWASP Foundation* [OWAb]. Countermeasure, such as *Compliance Testing* or *Compliance as Code*, exist and support to reveal *Environment* misconfiguration. However, the *designated* usage of the *Environment* is not tackled, i.e., operated applications are unlikely to be considered nor *explicitly* tested for configuration compliance.

To support software vendors in defining, creating, modeling, executing and evaluating compliance requirements of their products against their operating *Environments*, we conduct research on *Compliance Testing* from a *product-point-of-view*.

We propose a generally applicable, incremental and continuous process model, i.e., *Continuous Compliance*, to determine the process of crafting compliance requirements in the context of product (software) related configuration. Furthermore, we present a more practical, tool-supported extension of this process, i.e., *Continuous Compliance Testing*, to enable a straightforward adaptation of this new approach.

Pursuing a model-driven approach of creating *Compliance* for a modeled *Product* and the usage of *Data-Driven-Testing* principles, we facilitate a high *Test* reusability and *context-sensitive* parametrization.

Finally, with our *Compliance Management Tooling*, we create the opportunity to *define*, *inspect* and *evaluate* product-centric compliance requirements across different *Environments* and reveal issues with system or product specific configurations.

Contents

List of Figures

1. Introduction

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

ANONYMOUS

Contents

This thesis is primarily motivated by the technical debt of security aspects, which arise due to increasing software and environment complexity. Besides the challenges of growing pace in software development, e.g., the automation process of different stages, such as building, testing, and deployment, to continuously provide reproducible artifacts [Hue12], the utilized environment and its configuration management becomes a considerable aspect [Wat99].

As stated in the *Top 10 Application Security Risks* report for by the *OWASP Foundation* [OWAb] and outlined by *Tenable* [Gar15], a well-known vendor for security software, *System Misconfiguration* is a crucial issue for today's environment. A missing accurate system configuration leading to security issues strengthens the assertion of technical debt regarding security aspects.

To assess this problem and to ensure accurate configuration, first attempts such as *Compliance Testing* and *Compliance as Code* exist. Adopting these concepts lets a tester utilize *Compliance* standards which are technically ensured through *Whiteboxtest* tests. However, current strategies in the field of *Compliance Testing* do not face the designated use of the *Environment*, i.e., considering the concrete operated application, nor do approaches or tooling exist which facilitate or ensure a product related system configuration. For this very reason, we establish a guideline on how to design and ensure *Compliance* of system configuration from a *product-point-of-view.*

To further motivate this thesis, we present a more detailed overview on our intention in Section 1.1. In Section 1.2, we outline our main contributions and in the last Section 1.3, we give a brief overview about the upcoming chapters.

1.1. Motivation

To extend the sight of an *Environment* while facing system configuration, i.e., considering concrete *Product* related system configuration requirements, we presented the approach of *Compliance Testing* from a *product-point-of-view*. We intended to model *Compliance* independently from the executing system, i.e., the *Environment*, which is acceptable since the accurate configuration arises from the operated *Product*.

We present a novel process model (*Continuous Compliance*) to clarify all activities required to derive *Compliance* from a *product-point-of-view*. This process is based on a specially crafted domain (cf. Section 5.1), which allow the separation of a *Product* its *Compliance* and the required *Tests* needed to ensure compliance. To this end, we adopt the principle of *separation of concerns* and indirectly yield reusability. Furthermore, a software vendor should obtain the possibility to define *Compliance* once and reuse them to describe environment related requirements for other *Products*.

Given an *Environment* independent modeling of *Compliance* requirements, we further attempt to tackle the separation of *Tests* and the accurate *Test Data*, which is derived from its context, i.e., the operating system, during test execution. For example, certain configuration requirements could vary between the operating *Environments*, e.g., on one environment the log directory is located under `/var/log/` whereas another *Environment* uses `/opt/applications/log/`. To support this variation, we follow the principles of *Data Driven Testing* and intend to represent the actual *System Under Test (SUT)* as *ApplicationSystem*, i.e., the combination of *Product* and *Environment*.

Our research attempt should lead to a solution which allows the creation of vendor specific compliance and should be applicable in different contexts, e.g., on different *Environments*. Additionally, such *Product* related *Compliance* should yield the ability to establish quality gates as part of a *Continuous Delivery* pipeline. For example, the operating environment which operates a product could be checked.

All contributions are explained in more detail in the next section.

1.2. Contributions

The overall goal of this thesis is to submit a process model blueprint and to develop a domain-driven software concept to tackle the non-existing ability to define, model, and execute compliance requirements from a *product-point-of-view*. Therefore, we outline our two main contributions of this thesis, which lead to the final contribution.

The first contribution of this work is a process model blueprint (*Continuous Compliance*) which outlines three central phases, i.e., *Elaborate*, *Develop* and *Evaluate*. We describe required roles together with their responsibilities and relate them to the created process model. Besides, we present a mapping of our phases and activities to an existing incremental model, i.e., the *Software Development Life Cycle (SDLC)*.

Our second contribution is a more technical motivated process, based on *Continuous Compliance*, to promote its practical adaptation, i.e., *Continuous Compliance Testing* yielding a tool support of this process. We present a novel black-box framework realization which allows modeling and execution of *Compliance* against (customer-) *Environments*, called *Compliance Management Tooling.* To reach a high reusability of already implemented compliance test, we present a concept for *Test Genericity* on test code level. Such generic test are then instantiated during compliance test execution in by means of its context, i.e., the *ApplicationSystem.* We discuss, evaluate, and compare our concept against exiting solutions, such as Chef Automate, to clearly state the difference and innovations of our approach. Additionally, we conduct a case study with a software vendor partner to evaluate our tooling approach against their needs. We collect and analyze their received feedback and propose further improvements and future work.

To summarize, this thesis yields two significant contributions: a novel process model for *Continuous Compliance* and a tool support to facilitate the adaptation of *Continuous Compliance Testing.*

1.3. Structure of this Thesis

First, we introduce all relevant background information in Chapter 2 needed to comprehend our proposed solution. Starting with basic information and techniques on compliance testing to support the fundamental idea. We introduce software engineering related topics, such as *SDLC, RUP*, and *CMM*, which we use to classify and evaluate our concept, afterward.

In Chapter 3, we formulate and explain the central problems which we are tackling with this thesis and explain their origin in detail.

To isolate our proposed solution from exiting approaches on security/compliance testing and infrastructure compliance automation, we present *Related Work* in Chapter 4. We focus on the three main fields of related work: *Security Engineering*, *Compliance as Code*, and *Infrastructure Compliance Automation.* For each field, we introduce, discuss, and clarify their problems concerning software related compliance modeling.

The novel process model, i.e., *Continuous Compliance*, presented in Chapter 5. First we establish the overall domain model in Section 5.1. Afterward, we present required roles, responsibilities, and a mapping towards *SDLC.*

To promote the practical adaptation of our process model, we present the approach of *Continuous Compliance Testing* in Chapter 6. First, we present a rough mapping towards the process model and define technical aspects in more detail as part of the software requirement specification in Section 6.2. Based on this, we outline a meta model to allow modeling of our particular domain and present further, more technical, concepts to facilitate *Resuability* (cf. Section 6.4.1) and enable *Reporting* (cf. Section 6.4.3) of collected results.

In Chapter 7, we present a blueprint software architecture to obtain the novel blackbox framework to accomplish *Continuous Compliance Testing.* We outline detailed aspects of the implementation, applied design patterns, used tool and frameworks. Finally, we describe our proposed *Command Line Interface (CLI)* (*cmt-cli*) to support remote execution and present the realization of the previously defined *Reporting* concept.

We evaluate our presented approaches, i.e., the *Continuous Compliance* process, the

Compliance Management Tooling, and the concept of *Reusability*, based on a case study in Chapter 8. More precisely, we present and discuss the results of a case study enrolled at KISTERS in Aachen. Furthermore, we mention problems and pitfalls we struggled during our evaluation.

Finally, we conclude our work and propose possible future work and improvements Chapter 9.

2. Background

Don't panic!

Douglas Adams

Contents

In the following, we introduce and explain the required background.

The first sections *Software Testing* (Section 2.1), and *Compliance Testing* (Section 2.2) introduce the fundamental testing idea and the objective under test, namely the infrastructure. These foundations are essential to comprehend the *Problem Statement* (Chapter 3) and the *Related Work* (Chapter 4) on which this thesis builds upon.

Afterward, in Section 2.3, we introduce variants of the *Software Development Life Cycle (SDLC)*, which are elements of the presented concept (Chapter 5) and fundamental of the concept classification as part for the evaluation (Chapter 8).

Finally, we conclude this chapter by explaining the concept and idea of *Capability Maturity Models* (Section 2.4). Variants of this model are used in the conclusive Chapter 8 covering the evaluation and classification of our presented solutions.

2.1. Software Testing

In the field of dynamic software testing as part of the *Quality Assurance (QA)*, two main approaches exist which represent different actions for test case derivation. However, since we mainly focus on the field of compliance testing, we only present a brief classification and outline differences between Whitebox and Blackbox testing.

2.1.1. Whitebox Testing

Whitebox or *Glassbox* testing focuses on the internal structure and states of function, objects or systems. Thus, a tester needs to have full access to the *Unit under test (UUT)*, i.e., its source code. Before applying a whitebox testing approach, the tester needs to have a significant understanding of the software architecture and has to use her knowledge about the internal structure to design the test cases [PM04]. The most common whitebox testing approaches are *State-Coverage* testing and *Data-Flow* testing [LL13].

2.1.2. Blackbox Testing

Blackbox testing, also known as *Functional* testing, focuses on the action and its result of a function or system under test. It refers to analyzing a running UUT by probating with its inputs [PM04]. Therefore, a tester does not need to know the concrete structure of the UUT. The only required knowledge to design a Blackbox test case is the expected behavior (result) of a function or method which can depend on certain input values. To obtain those parameters, the requirement specification or other related documents can be conducted [LL13]. Finally, a test simply simulates the functionality using the pre-defined input values and compares the received result to the expected one. The best know approach, based on the Blackbox testing strategy, are *Equivalence-Class* testing and *Boundary-Value* testing [LL13].

2.1.3. Data Driven Testing

Data-Driven testing (DDT) describes the approach of testing a UUT or SUT using a predefined set of *test data*. As testing is often a repetitious task, one might be interested on running essentially the same test with slight different system inputs and verify that the actual output varies accordingly [Mes07]. This ability is made possible with the approach of DDT.

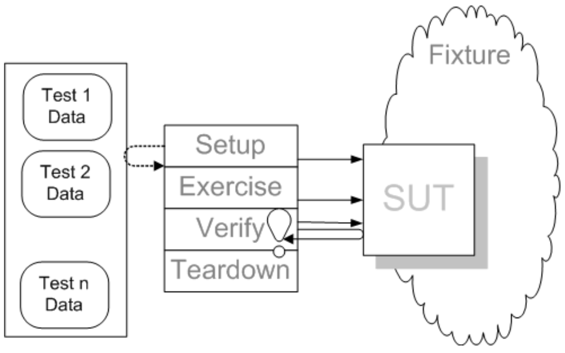


Figure 2.1.: Exemplary illustration of the Data-Driven-Testing approach [Mes07].

As shown in Figure 2.1 the identically test run, i.e., *Setup*, *Execute*, *Verify* and *Teardown*, is take for each input data. The used test data for each run is provided by an data set yielding an Input and Verification set.

To this end the approach of *Data-Driven-Testing* leads to the concept if *parametrized testing* which we use in our upcoming concept to support *test reusability.*

2.2. Compliance Testing

Compliance Testing, also known as *Conformance Testing*, *Regulation Testing* or *Standards Testing*, is part of the non-functional software testing field [Int; Sof; Tec]. It is used to test and to verify that given standards in form of *non-functional* requirements are fulfilled by the software itself, the development process, or the system hosting this software. The phase in which the test should be applied is defined by the standard, e.g., the *CIS Benchmarks* on infrastructure compliance should be preformed between the build and deployment phase. To this end, *Compliance Testing* can be performed to ensure that deliverable artifacts are in a compliant state during each phase of the development life cycle [Sof].

Due to legislation enhancement, e.g., the data security standard for Banking and Financial Service *Payment Card Industry Data Security Standard (PCI-DSS)* [Cou], the relevance of compliance testing and its amount of application increases. *Compliance Testing* supports a software vendor to certify certain standards, e.g., PCI-DSS, ISO-27001[1] or ITIL[2]. Alternatively, it enables to adhere on standards recommended by appropriate institutions, such as BSI and CSI, to ensure infrastructure quality in general.

Particularly in the area of security conformance testing, the amount of existing standards which could be applied and certified is high. To derive a fundamental baseline testing set, we explain some of them in more detail in the following. Afterward, we introduce a relatively new approach, called *Compliance as Code*, which allows compliance testing in an automated manner.

Finally, we report that the fundamental testing approach – Compliance Testing – is a blackbox testing approach as well, since we are not interested in the realization but rather in the correct configuration of UUT. However, on a higher level of abstraction, i.e., focusing on the infrastructure and not the software configurations, one would perform a whitebox test, because the environment itself and not its configuration is tested. This situation allows us to execute more fine-granular tests on the UUT, because one is able to verify significant more configuration settings from the inside rather than guess-checking them from the outside, e.g. set of users which are allowed to login via SSH.

Definition For this thesis and its related domains and concepts we defined the term of *Compliance* as set of precisely formulated definition of infrastructure demands to ensure correct system configuration, i.e., to reveal system misconfiguration. A *Compliance requirement* denotes one item of this set. In short, we always refer to infrastructure configuration. For example a *compliant infrastructure* fulfills all its demand configuration, i.e., those defined due to its designated usage.

[1]`https://www.iso.org/isoiec-27001-information-security.html`
[2]`https://www.axelos.com/best-practice-solutions/itil/what-is-itil`

2.2.1. Benchmarks

In the field of compliance, regarding law appropriated standards or especially cybersecurity, different organizations offering benchmarks exists, e.g. *CIS* and *BSI*. These standards are a collection of best practices and recommendation for system and software configuration to gain as much security as possible. Furthermore, they can be classified as complex guidelines (requirement specifications) for compliance testing. Two very popular organizations offering security compliance standards are the *Center of Internet Security (CIS)* and *Federal Office for Information Security (BSI)* [Cen17; ISb].

CIS Benchmarks are available for download in electronic, textual form, i.e., PDF. They collect and describe various aspects for an objective under test, e.g., mobile device, network device or a server operating system, for security compliance (CIS Level) [IS17]. As part of a paid membership, the CIS also offers hardened images as well as remediations scripts to easily apply the proposed configuration. However, those guidelines are not part of our work.

BSI Grundschutz is published in textual form, collects and describes security configurations as well. Certainly, the BSI Grundschutz is much more complex and covers more general and different areas of interest for an IT Company compared to the CIS Benchmark. For example, the BSI provides guidelines for rescues and plans for natural hazards, such as fire. In the scope of this thesis, we only focus on subset of categories B3 (*Sicherheit der IT-Systeme*) and B5 (*Sicherheit in Anwendungen*).

Since these guidelines are designed for a group of different skilled people, i.e., from *Chief Security Officer (CSO)*, *Product Owner (PO)* up to the *Software Developer (SD)* and *Business Manager (BM)*, they are mostly written in informal language to be easily read- and understandable. However, this design decision makes them difficult to be implemented because no concrete test or checking method/script is given and linguistic failures, such as generalization or redemption, could lead to misinterpretation. Therefore, a satisfying and exact implementation of the required compliance rules quickly results in a difficult task. For an enhancement, *Compliance as Code* was published and tries to close this existing gap between the compliance awareness and realization.

2.2.2. Compliance as Code

Compliance as Code (CaC) can be seen as the process of building compliance into development and operations [Bir16a]. Even more, it brings different roles, which are involved into a software project, i.e., manager, internal audit, project management office, and developer, to “one table” to define compliance requirements and rules upfront [Bir16a]. The enhancement of CaC is the reduction of overhead and paperwork which is nowadays required during specification and definition of compliance requirements. The definition in form of code makes it easier to test a system for compliance since existing code can be executed, shared, and reused. Compared to already established approaches, such as *Compliance Management System (CMS)*, the approach of CaC attempts to improve the pace and opens the domain of compliance testing to other domain experts, such as developers or release managers. Furthermore, having executable compliance as code, enables automatic tests as part of a continuous delivery pipeline. The most up to date and important tool which supports a simple definition and execution of compliance as code is *InSpec*[3].

InSpec by Chef was founded back in 2016 and is based on the idea of the well-known Serverspec. However, it was fully (re-) written from scratch to improve the design and better usability [Che16b]. Compared to Serverspec, InSpec tries to add more resources for testing and implements its own domain specific language, i.e., *InSpec DSL*. Besides using the well-designed language, all InSpec tests could be written in plain ruby as well. Due to the easy read- and understandable DSL, InSpec tests also serve as a baseline documentation for compliance tests, which is required by most audit processes. Using InSpec, enables a tester to easily check the correct configuration of a resource. For example, Listing 2.1 shows a compliance test for a ssh-daemon which should be configured to listen on port 22.

```
describe sshd_config do
    its('Port') { should eq('22') }
end
```

Source Code 2.1: Sample InSpec source code ensuring that the SSH service port is configured to 22.

[3]https://www.inspec.io/

2.3. Software Development Life Cycle

The *Software Development Life Cycle* (SDLC) describes a unified process for planning, developing, testing, and deploying software or information systems. The life cycle tries to separate the overall development process into unique, distinct and chronological activities [Win05]. Each defines its preconditions and outcomes, which are used in the preceding activity. Overall, each life cycle starts with the phase of planning and analysis, i.e., collecting all customer requirements, and ends with the delivery [LL13].

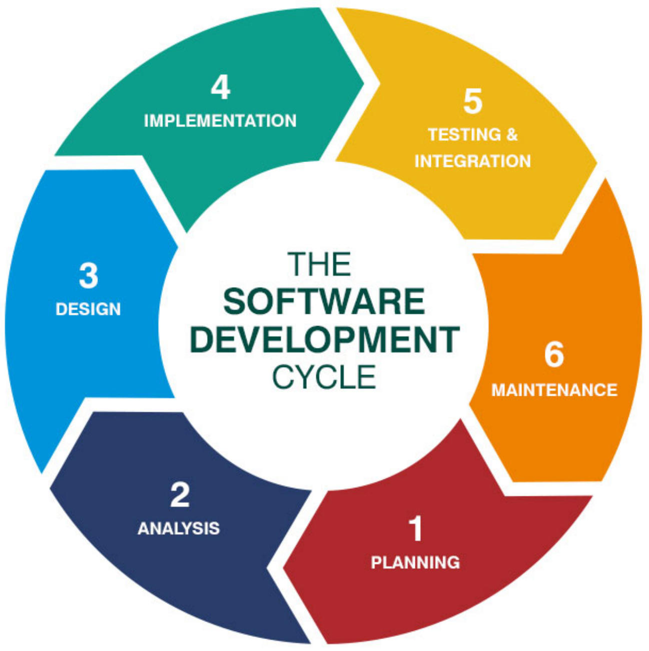


Figure 2.2.: Diagram illustrating all six relevant phases of the *Software Development Life Cycle* [Hus16].

A very common segmentation and flow is shown in Figure 2.2 and consists of six activities *Planning*, *Analysis*, *Design*, *Implementation*, *Test & Integration*, and *Maintenance*. Depending on the planned kind of usage, different and more concrete instances of this model exist. Agile methods, such as *Scrum* and *Extreme Programming (XP)*, incremental or iterative methods, such as *Rational Unified Process (RUP)* or “custom” extensions, i.e., with the aspect of *security*.

In the following, we introduce the RUP and a security-extended SDLC in more detail.

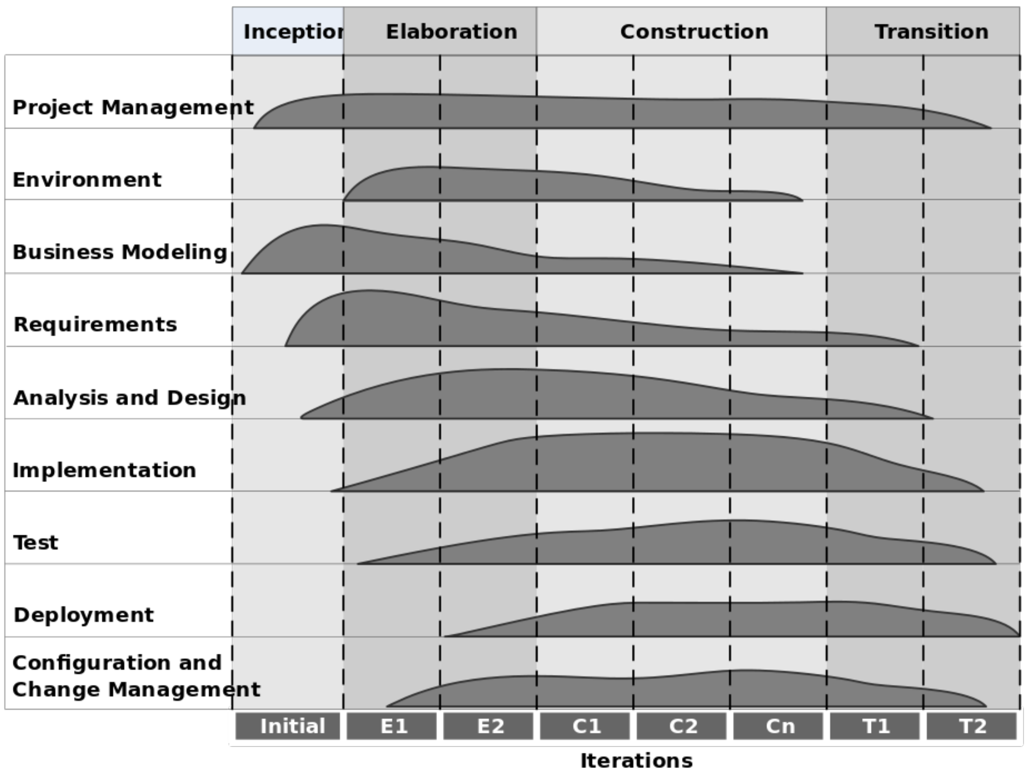


Figure 2.3.: General process illustration of the Rational Unified Process. The header shows the four main phases along with the different disciplines on the left side [GFL06]

Rational Unified Process Similar to the SDLC and many other process models, the RUP is separated in single chronological phases [Win05; LL13]. As shown in Figure 2.3, the RUP distinguishes between four main phases, i.e., *Inception*, *Elaboration*, *Construction*, and *Transition*. Each of those phases can be repeated iteratively, which justifies the classification as an iterative process model, and consists of a set of *core-workflows* as illustrated on the left side in Figure 2.3. These workflows can be grouped, more general workflows: *Requirements*, *Analysis*, *Design*, *Implementation* and *Test*, as described in the *Unified Process Model* [LL13]. The distribution for each core-workflow at each phase indicates the time effort which should be spend on this desired task. E.g., during the *Initial Iteration* and during the *Inception Phase*, one should spent the most time effort on the *Business Modeling* (cf. Figure 2.3).

Due to its iterative basis and separation in different phases, the RUP is suitable to support a continuous and repeatable process [Win05].

Security in SDLC Lately, two organizations, namely Microsoft and the *Open Web Application Security Project (OWASP)*, presented an SDLC extended with different aspect concerning secure software development since the importance of secure software development increases. Both propose such a process models to encourage an increased and early application of security-relevant actions as well as to reduce the amount of security issues [OWA17b].

In 2004, Mircosoft proposed the *Security Development Lifecycle (SDL)* to address security compliance requirements during software development [Mic04]. SDL is very similar to the basic SDLC model with two additional phases. At the very beginning, SDL describes a *Training* phase which should be enrolled to train all involved persons on core security topics, e.g., threat modeling or secure coding. Furthermore, it introduces the *Verification* phase after the *Implementation* phase. This phase extends the standard *Testing* phase with additional security-related testing disciplines, such as *Attack Surface Review* [Mic04].

The *Secure Software Development Life Cycle (S-SDLC)* by the OWASP Foundation [OWA17b] – which is still in development – presents a guideline about how to integrate and combine any SDLC process model with their *Security Assure Maturity Model* (OpenSAMM, cf. Section 2.4). The S-SDLC, therefore, defines security software development process as well as guides, tools, checklists, and templates of activities in each phase [OWA17b]. To this end, each step of the fundamental SDLC process model is extended with particularly security related activities. For example, the common *Design* phase is renamed to *Security Design* and additionally conducts the discipline of *Application Threat Modeling*. Alongside with the enlargement of typical phases, new phases at the beginning of the SDLC are added, i.e., *Overall Flow* and *Security Awareness Training* which are similar to the initial phase in the SDL.

2.4. Capability Maturity Model

Capability Maturity Model (CMM) is a development model in the field of software development which aims to improve and to rate existing software development process [Pau93]. The best known derivate of this model is the *Capability Maturity Model Integration (CMMI)*, which defines five different level of maturity, i.e., *Initial* (Level 1), *Repeatable* (Level 2), *Defined* (Level 3), *Managed (Capable)* (Level 4), and *Optimizing* (Level 5) [Pau93]. Each level outlines different aspects on how an individual aspect of the software development process, e.g., Project Management (Level 1), should be implemented and realized. To be adopted, each level requires the previous level to be entirely implemented as they mostly build upon the previously required aspects.

Besides this well-known derivate (CMMI), various other maturity models in the area around software development and its process exist. Particularly in the field of security, some models gain significant publicity since most vendors are interested in proving that they stick to certain, well established, security practices during their software development.

Open Software Assurance Maturity Model The *Open Software Assurance Maturity Model (OpenSAMM)* defines itself as open framework to help organizations to formulate and to implement a strategy for software security [OWA17a]. Similar to CMMI, OpenSAMM is split into different Business Functions as illustrated in Figure 2.4. However, theses four fields, i.e., *Governance*, *Construction*, *Verification*, and *Deployment*, are independent from each other and can be realized separately as compared to CMMI.

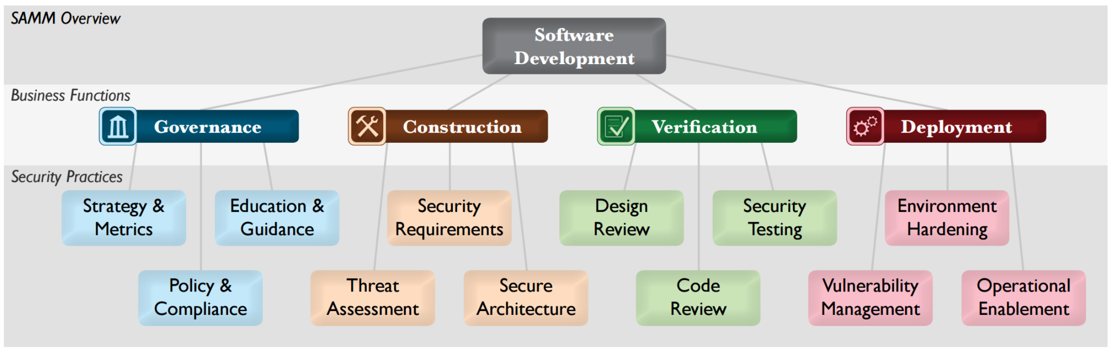


Figure 2.4.: OpenSAMM overview on Business Functions and Security Practices [OWAa]

Each *Security Practices* relates to one *Business Function* and is then split into three different sub-levels (Level 1-3) which describe concrete objects and activities to reach a certain degree of security on the chosen practices. For example, the security practice *Environment Hardening (EH)* as part of the *Operations* business function (cf. Figure 2.4) is split into levels *EH1 - EH3* (EH1 - *Baseline Operations*, EH2 - *Applications Operations* and EH3 - *Application Health*). The objective on every level varies and requirements increase the higher the level gets (cf. [OWAa]). E.g., EH1 focuses on understanding the baselines of the operational environment, whereas EH3 requires validating application health and status against known best practices. To adopt or reach these objectives, short and helpful descriptions of the related actives are given, e.g., to obtain E1 Level one has to “Maintain operational environment specification” and to “Identify and install critical security upgrades and patches” [OWAa].

Based on this theoretical foundations, we continue with the central *Problem Statement* and its detailed source of origin in Chapter 3.

3. Problem Statement

Indicated trough the *OWASP Top 10 Application Security Risks* [OWAb] and moreover endorsed by *Tenable* [Gar15], *System Misconfiguration* is a crucial issue leading to security vulnerabilities. Approaches, such as *Compliance Testing* or *Compliance as Code*, exist and support the discovery of flawed configuration settings of an environment. Adapting tools, such as Chef InSpec [Che16b], UpGuard [UpG], or ServerSpec [Gos13], combined with available security guidelines, such as the *CIS Benchmark* [IS17] or the *BSI catalog of measures for hard- and software security* [ISb], the overall effort and process of compliance testing is very straightforward and well-elaborated.

However, nowadays none of these approaches covers creating and testing of product related configuration which seems insufficient since the designated usage of an *Environment* has to be somehow conducted. The relevance of such a *product-point-of-view* on system configuration settings was motivated in Section 1.1. Although defining *Compliance* for once specific *Environment* (single) is very straightforward, enlarging the scope towards *Environment* independent representation increases the difficulty, i.e., reusing and altering *Compliance* to fit the actual context (*ApplicationSystem*). Hence, a modular approach of defining, assigning and testing has to be elaborated to reduce the overall complexity of product-related compliance testing. As a result, the extending view, i.e., tackling product-related configuration, on *Compliance Testing* creates the central set of problems.

Additionally, integrating product-related configuration requirements becomes questionable, i.e., *Compliance* form a *product-point-of-view*, into the already established process of *Compliance Testing*. Furthermore, how to craft and document all resulting *Compliance* demands related to its origin, i.e., the *Product*, has to be considered. The previously described set of problems is categorized into three general sub issues, i.e., *System Misconfiguration*, *Information Magnification* and *Management Inexperience*. The concrete problem and their relation between each other is outlined in the following.

System Misconfiguration First and foremost, *System Misconfiguration* is one high risk with respect to system security nowadays. Since large software vendors and developers often rely on external dependencies or use multiple, smaller, own software components simultaneously, a complex software is quickly created. This situation leads to, mostly, unconsciously created security vulnerabilities [OWAb]. Especially large software projects, consisting of multiple vendor specific components, require proper configuration and system settings. Each of those components mostly relies on certain system functionality to provide its service. To check an existing system for compliance, concerning the required configuration and prerequisites, one needs to have the possibility to model and finally execute the software-related *Compliance* requirements. This requirement indicates a central problem regarding system misconfiguration for software products, since a solution to model security related polices from a product-point-of-view is missing so far and not supported by existing compliance testing system (cf. Section 4.2) or *Infrastructure Compliance Automation* systems (cf. Section 4.3). The next problem occurs when trying to manage and to document the created compliance requirements for varying *Products*.

Information Management The problem of *Information Management* surfaces due to the huge amount of compliance requirements which need to be managed, versioned, and documented. Besides *System Misconfiguration*, this problem is likewise crucial since information management is the key point to facilitate reusability of already created documents, i.e., *Compliance*. Yet, no possibility exists which allows defining compliance requirements and reusing them for different *Products*. To enable a later audibility for certifications of various kinds (cf. Section 4.1), a repeatable, fully documented process is required.

Management Inexperience Besides the problem of how to model and manage compliance polices from a *product-point-of-view*, the persons and roles involved in this process have not been clearly identified. Due to the thematically relation between Security Testing and DevOps area, because Security Testing deals with certain aspects applied during development as well as operations (DevOps), certain roles are involved. For now, trying to realize compliance requirements for a specific software product, the approach will leads to obscurities in the field of responsibility.

This effect introduces the problem of *Management Inexperience* since no model exists which describes the allocation of responsibilities. Furthermore, as the field of *Compliance Testing* from a *product-point-of-view* is new, no concrete role assignment exists nor the required awareness is given. To define cooperate-like compliance requirements including all involved stakeholders is a necessity. It is to clarify which roles are required and how the responsibility for certain tasks – during compliance modeling – is mapped.

Scope

The scope of this thesis includes all three outlined problem statements. First, we propose an overall proceeding to create a baseline for *Compliance Testing* from a *product-point-of-view*, i.e., we implement a process model based on established procedure models. Additionally, to primarily tackle the problems of *Information Management* and *System Misconfiguration* we create a tooling. On the one hand, this tooling should support the user in fulling the implied tasks, i.e., defining, modeling, executing and documenting *Compliance*. On the other hand, it should allow a practical adaptation of the novel process model.

However, we deliberately do not intend to fully solve each outlined problem. For example, as part of *System Misconfiguration* we refrain from considering any actions concerning environment hardening or remediation processes, nor do to elaborate on further security related approaches, such as *Static Code Analysis* or *Security Scanning*. The main scope is *Security auditing* of predefined *Compliance* modeled from a *product-point-of-view*.

Being aware of these centric problems, their origin and possible options to tackle them, we present *Related Work* on the topic of *Compliance Testing* and discuss their opportunities and inadequacy in the next Chapter.

4. Related Work

Don't panic!

Douglas Adams

Contents

In the current field of research, the topics of *Compliance As Code* and *Continuous Compliance Testing* are relatively new and barely researched. Paper and literature about scope-extension/-transformation from *DevOps* to *SecDevOps*, *DevSecOps* or *DevOpsSec* exists [McG16; Bir16a; VO10; Sch15; Sto15] and all claim that security needs to be embedded into the current approaches of DevOps. To this extend, first attempts facing *Compliance As Code* emerged and enable security testing and compliance verification in an automated manner [Bir].

Tools, such as Chef InSpec[Che16b], RedHat OpenScap[Red08], and Nessus[Ten16], implement the idea of *Compliance As Code* and allow the definition as well as the execution of compliance tests. Unfortunately, these tools just cover the general definition of compliance rules mainly focusing on the infrastructure. However, they do not allow conjunction or other kinds of modeling to represent certain software product and their implied compliance running on specific infrastructure. Consequentially, a software vendor can not model its software related compliance requirements, i.e., from a *product-point-of-view*, and can not test those requirements in a continuous manner using existing tools and build pipelines.

In the following, we outline the prime related work in the fields of Security Engineering (Section 4.1), Compliance As Code (Section 4.2), and Infrastructure Compliance Automation (Section 4.3), since all these topics are in-cooperate with our presented work.

4.1. Security Engineering

The scope of Security Engineering focuses on different aspects, such as tools or processes, and methods required to design, implement, and test complete systems. One central, but not sufficient, required skill is proper Software/System Engineering, i.e., business process analysis, evaluation, and testing [And10]. Thus, we focus on related topics in this area and argue their relevance for our proposed solution.

4.1.1. DevOps and Security

The interaction of DevOps and Security is a relatively new intermediate-discipline describing how to combine *Security* and *DevOps*. Already in 2015, Storms [Sto15] discussed and outlined approaches on how to integrate the aspect of security in the process of agile software development and DevOps. In 2016, Mohan et al. [MO16] examined the abbreviation SecDevOps and showed how to combine Security and DevOps. They outlined that the most need for security in DevOps is raised through the lack of compliance concerns. The most recent publication is by Jim Bird [Bir16b] where he describes DevOpsSec and how to secure the whole development process while adopting security to DevOps.

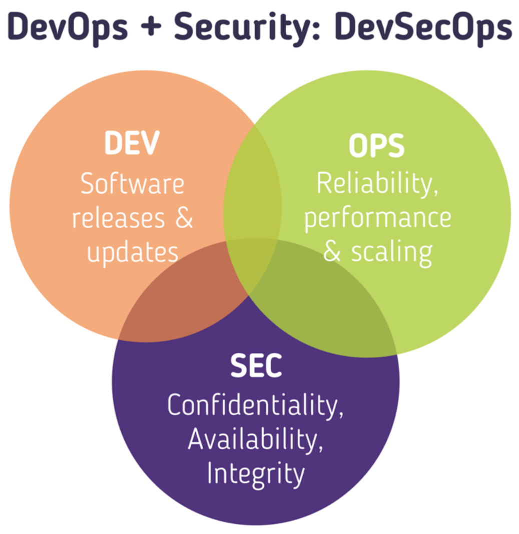

DevOps + Security: DevSecOps
DEV
Software releases & updates
OPS
Reliability, performance & scaling
SEC
Confidentiality, Availability, Integrity

Figure 4.1.: Conjunction of all three different disciplines and its activities, resulting in the new intermediate-discipline DevOpsSec [Von16]

The uncertainty regarding the correct term, e.g., may it be SecDevOps, DevSecOps or DevOpsSec, is sufficient discussed and depends on the moment when to use security in a DevOps process. As termed by Stroms [Sto15], Bird [Bir16b], and Tischart [Tis], we also rely on DevOpsSec. This, means that Security is the last step in the overall process.

We have shown that the approach of integrating security into DevOps (DevOpsSec) is not new and well researched. Thus, introducing and linking compliance requirements against software artifacts in an early stage of development, e.g., to receive security tests at the end of development, is acceptable.

4.1.2. Benchmarks

In the discipline of security engineering, reporting and risk calculation is an important part [Jan10]. It allows the manager or any other responsible person to classify and estimate the current security risks. To gain a first fundamental classification, various standards exist. These standards are often used to implement and reach a minimal essential security of a system or process. The most common institutions proposing such standards are Center for Internet Security (CIS) and Federal Office of Information Security (BSI) as stated in Section 2.2.

The CIS proposes many different standards for various fields and each condenses a set of best practices and known security flaws which need to be checked. For example, benchmarks for Linux, Windows, or Mac operating systems (OS) alongside with mobile OS's, such as Android and Apple iOS, are available for download [Cen17].

In the field of compliance testing from a *product-point-of-view*, evaluating the hosting system (operating system) is necessary. Thus, to compile a proper, very straightforward set of *baseline tests*, one should rely on the existing CIS Benchmarks for Windows and Linux. The CIS Benchmark for *Microsoft Windows Desktops*[1] and *Distribution Independent Linux* [2] provide essential documentation on how to check and remediate "classic" and well-known Windows or Linux security flaws. We use both, primarily the Windows 10, during the evaluation of our proposed solution to create a fundamental (compliance) *baseline test* which is then extended or softened due to product specific requirements.

4.2. Compliance As Code

To test a system or infrastructure for compliance in an automated manner, i.e., during deployment, one should rely on *Compliance As Code*. As explained in Section 2.2, this approach offers various advantages, which is why we also rely on existing standards in this field. The most public standards in this area are briefly outlined in the following.

4.2.1. SCAP

The *Security Content Automation Protocol (SCAP)*, proposed by the National Institute of Standards and Technology in 2006 [ST09], currently exists in four major versions (1.0 - 1.3). The basic idea of SCAP is to combine various (existing) standards used to test for software security flaws and configuration issues. Since version 1.0, the *eXtensible*

[1] `https://www.cisecurity.org/benchmark/microsoft_windows_desktop`
[2] `https://www.cisecurity.org/benchmark/distribution_independent_linux`

Configuration Checklist Description Format (XCCDF) and the *Open Vulnerability and Assessment Language (OVAL)* (OVAL) languages are supported [ST09]. Compared to other standards, SCAP is one of the few specifications, which is officially accepted and approved by the National Institute of Standards and Technology.

The XML based specification, due to the usage of XCCDF and OVAL, allows modularity and reusability of existing rules. This characteristic enables a generative approach of test case derivation, and is the reason why we suggest SCAP as one standard to be used in combination with our upcoming solutions.

4.2.2. Chef InSpec

Chef InSpec is a relatively new approach in the field of Compliance As Code, as stated in Section 2.2. Its tests are written in a separate domain specific language (InSpec DSL), hence so far no huge community exists and all existing tests, e.g. (Open)SCAP definitions - which are far older - need to be rewritten when adopted.

The approach introduced by Chef[3], i.e., InSpec DSL, focuses on high combination of already existing compliance profiles and is realized by *Meta Profiles* [Har17]. For example, an existing profile checking the Windows patch level, which is public available on GitHub[4], can be easily adopted in a separate test suite. Besides external dependencies inclusion, local test can be used as well [Har17].

These capabilities, i.e., inclusion and dependencies management, are ideal when following a generative and modular test case design and justify the usage and integration in our compliance management tooling.

4.3. Infrastructure Compliance Automation

With the help of *Infrastructure Compliance Automation (IaC)*, automatically checking an infrastructure for compliance is possible. As the popularity of *Infrastructure as Code (IaC)* increased in the past years, compliance automation was not neglected, and various tools/scripts were presented, which allow more or less Infrastructure Compliance Automation.

In the following, we outline the most important tools in this field. *Chef Automate*, *OpenSCAP by RedHat* and *Nesuss* should be mentioned beforehand since they are the most well known-open source tools (compared through GitHub ranking). We briefly describe their functionality and argue to which extend these tools could or could not solve or reduce the outlined problem. This listing raises no pretense of completenesses, is ordered alphabetically, and represents the result of our research at the current time.

[3]Vendor of InSpec - `http://www.chef.io`

[4]`https://github.com/dev-sec/windows-patch-benchmark`

Chef Automate is a full-featured enterprise service based on Chef InSpec to manage, deploy, and verify infrastructure defined as IaC using Chef [Che16a]. Using Chef Automate, one can continuously test its infrastructure against a set of defined InSpec Rules and therefore check it for compliance regarding these rules. Since the vendor Chef provides both tools used alongside with Chef Automate itself, namely Chef and InSpec, the integration works well.

Chef Automate can be used to check an Infrastructure for existing *System Misconfiguration*, since it fully relies on InSpec, whereas this problem (cf. Section 3) can be solved. The other two, main, problems are not tackled yet. On the one hand, modeling the utilization of one or more infrastructure is not supported (*Information Management*), i.e., the *product-point-of-view*, and on the other hand *Management Inexperience* is not fully solved as well. The management (stockholder) is supported through various reports, but due to the missing generality and abstraction, the management can not yet handle change or compliance requirements well.

Docker Bench for Security is a set of bash scripts which are designed for Docker containers to check them for compliance. The implemented compliance tests are aligned with the tests proposed by *CIS for Docker Container Security* [Doc16; Cen16]. However, *Docker Bench Security* is not meant to be used as continuous compliance testing, but one could argue that it is easily automatable through build-pipeplines and pre-deployment phases, i.e., Jenkins [Cir16]. This conjunction would help to tackle our identified problem of *Security Misconfiguration* but would overlook the other two problems (*Information Management*, *Management Inexperience*).

Greenbone and OpenVAS are vulnerability scanner based on the OpenVAS Framework [Com09]. Greenbone represents the Enterprise Version of OpenVAS and offers additional (paid) customer support. Both tools can run black- and whitebox tests on a system under test. The latter one enables these tools to be used for compliance testing and field tests defined through NVT, SCAP, or similar formats.

Greenbone and OpenVAS can continuously check SUTs for compliance, which solves the problem of *Security Misconfiguration* in some way. Through its built-in reporting and monitoring capability, Greenbone and OpenVAS can reduce the *Management Inexperience* and support user with information. However one can not use these systems to model a software-landscape which is operating on a given host whereas the problem of *System Misconfiguration* and *Information Management* are not yet fully solved.

Nessus is a top-rated and widely used software for vulnerability scanning founded by tenable [Ten16]. Due to its general functionality (scanning) Nessus is mostly categorized as blackbox testing tool since it scans the infrastructure (SUT) from the outside. But Tenable has discovered the growing trend in compliance testing from a security point of view and has extended Nessus (version 5 and higher) with a new plugin mechanism which allows running additional tests on the system site (whitebox) as stated in their report from January 2017 [Ten17]. Using one or more of the client-specific plugins, Nessus supports compliance tests as well.

Nessus compliance tests can be based on the standard XCCDF/SCAP format, which allows a high reusability of existing standards and compliance tests [Ten14]. However, by its nature, Nessus is not designed for extensive compliance testing whereas the stated problems can not or only be partially solved. *System Misconfiguration* can be checked but not modeled from a product-point-of-view. For the same reason, the problem of *Information Management* could not be tackled. Even more, since Nessus is a vulnerability scanner, no guidelines yet exists which outlines a process required to be applied by management to allow compliance testing from a *product-point-of-view.*

OpenSCAP is a toolset family founded by RedHat and mainly focuses on security compliance and vulnerability assessment [Red08]. The main tools presented by OpenSCAP are *OpenSCAP Base*, *OpenSCAP Daemon*, *SCAP Workbench*, *SCAP-Timony*, and *OSCAP Anaconda Add-on.* As the naming of this product family reveals, the fundamental testing framework (*OpenSCAP Base*) relies on the public, NIST certified, compliance testing standard SCAP (cf. Section 2.2).

With existing and self-defined XCCDF/OVAL benchmarks and test definitions, the detection of a *System Misconfiguration* is possible. Nevertheless, the generic modeling aspect is missing and the dynamic derivation of compliance requirements from a product-point-of-view are not yet respected. OpenScap in its fundamental functionally only focuses compliance testing from an infrastructure-point-of-view.

Rudder is a software founded by Normation in 2011 to run compliance tests against a given SUT and create to a descriptive report on a regular, e.g., daily, basis [Nor11]. Additionally, it supports continuous configuration and allows auto-healing of a given node when one or multiple tests fail. However, Rudder is not yet well known (cf. GitHub Ranking compared to InSpec) by which the tester/expert has to write Techniques (concept naming for test) manually with the help of generic, predefined test-pattern and can not rely on a huge community background. Rudder supports system- and user- defined variables which allows users to write more generic test in general. These variables allow a reuse of information system wide [Nor16], but can not be overwritten by hierarchy or a particular technique. However, the idea of Rudder's variables is not comparable to our concept of *Test Genericity* (cf. Section 6.4.1) since no redefinition or conjunction of values is supported.

With its massive support regarding technique modeling and continuous configuration (auto healing), Rudder solves the problem of *System Misconfiguration.* Additionally, with the support of great reporting capabilities Rudder tries to break down the necessary information required by management as much as possible and therefore lessen the Problem of *Management Inexperience.* Nevertheless, the issue of *Information Management* remains unsolved since (so far) no modeling of compliance requirements from a *product-point-of-view* is possible.

Security Monkey is a tool, presented by Netflix in 2014, to automatically monitor various cloud service instances for policy or configuration changes [NSM14]. It primarily focuses on plain monitoring, i.e., analysis and reporting. To this end, whitebox tests are executed on the hosting system (system under tests), whereas providing credentials is necessary. Apart from the analysis, Security Monkey stores the previous state of a policy or configuration and reports a warning to a specific user [Net14]. Due to its architecture, Security Monkey can be extended, which allows further implementation of custom security tests, i.e., a realization of technical *BSI Grundschutz* [5] requirements. Moreover, Security Monkey is running in a *continuous* manner whereby it can be classified as some kind of *Continuous Compliance Testing* Tool.

Compared to the previously outlined problem statement, Security Monkey can help to tackle the problem *Security Misconfiguration* since it can detect false configuration parameters. However, *Information Management* can not be solved using Security Monkey, because one is not able to precisely model the system's configuration based on previously created compliance rules or similar. Netflix's Security Monkey only checks all linked hosts against a given set of auditors or watchers [Net14]. Dynamic test generation based on placeholders is not yet supported, hence, correctly handling multiple systems is impossible.

Serverspec is the fundamental idea behind Chef InSpec, which is the core of Chef Automate as described in Section 4.3. The functional scope of Serverspec is similar to InSpec and allows the tester to run own or predefined tests on the SUT [Gos13].

We have considerably outlined the related work in the field of *Compliance Testing*. Furthermore, we have presented related tools and clearly outlined why each of them is not fully capable in tackling the present set of problems.

Based on these foundations, we propose our solutions to entirely tackle the existing problems in Chapter 5.

[5] https://www.bsi.bund.de/EN/Topics/ITGrundschutz/itgrundschutz_node.html

5. Conceptional Foundation

There are no facts,
only interpretations.

Friedrich Nietzsche

Contents

In this chapter, we outline a process-oriented and tool supported-concept to solve the central problem statement, i.e., modeling compliance from a *product-point-of-view.*

First and overall, we briefly introduce the elementary domain which we apply in all later concepts and realizations. Afterward, in Section 5.2, we present our applicable process model (*Continuous Compliance*) which creates the overall context for various, more fine-grained, activities and concepts.

5.1. Domain Model

To sketch our fundamental and viral domain for *Continuous Compliance* as well as *Continuous Compliance Testing* (cf. Chapter 6), we present central aspects and their relation in Figure 5.1. This domain model comprises all necessary aspects required to precisely define and introduce our overall domain and its terminology.

The most crucial part of our domain is the partition of three distinct aspects *Product*, *Compliance*, and *Test* (marked with a blue square in Figure 5.1) which express the fundamental modeling domain, i.e., the way how we intend to model compliance requirements from a *product-point-of-view.* We want to describe each compliance requirement for an *ApplicatioSystem* (the SUT) – a combination of *Infrastructure* and *Product* – by the use of a *Product* in the first place. A single *Product* is then capable of *demanding* certain *Compliance.* In a final modeling step, each *Compliance* can be *implemented* by the usage of a functional *Test.* This segmentation yields the central domain of our further concepts. To state each aspect and its purpose, we describe them in more detail in the following.

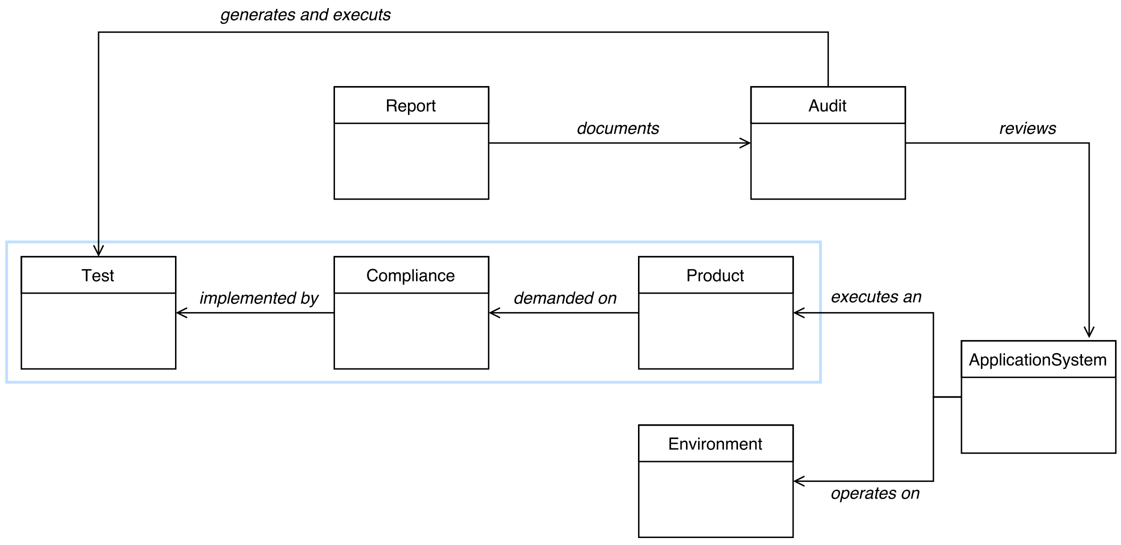


Figure 5.1.: The fundamental domain model defining the central aspects for modeling and describing compliance requirement from a *product-point-of-view.* It emphasizes the crucial partition of *Product*, *Compliance*, and *Test* through a blue rectangle.

ApplicationSystem describes the final SUT and moreover the concrete *context* for each modeled product and its compliance requirements. An *ApplicationSystem* is the conjunction of an *Environment*, e.g., a bare-metal server or virtual box, and the *Product* which is executed on this system.

This aspect allows further abstraction between the modeled product and its operating system. To this end, any product and its compliance requirements can be modeled in a more general way and rely on later, concrete, instantiation by means of *Environment* information.

Thus, when referring to a *concrete context* or *SUT* for a *Product* and its related compliance requirements or tests, we implicitly reference an *ApplicationSystem.*

Environment represents the system used to operate a product, i.e., the machine and its operating system. An *Environment* is required to model and describe the concrete context used to instantiate the (abstract) modeled compliance requirements, i.e., the *ApplicationSystem.*

Product is the most central point of this domain model. It is used to model a specific software and yields the "entry-point" to model or derive (from an existing model) all *Compliance* requirements and *Tests.* Furthermore, a *Product demands on* various *Compliance* which needs to be satisfied by the *executing ApplicationSystem.* To formulate these requirements, a *Product* it is associated with a *Compliance* element.

By means of a *Product* we introduce an additional modeling layer and apply the *product-point-of-view* on top of the *Compliance* aspect. To this end, we decouple the aspects of *Compliance* and *Infrastructure* as CaC usually does.

Eventually, a *Artifact* (cf. Section 6.3) is referred to as a *Product* as part of the presented *Software Meta-Model*.

Compliance describes a certain *standard* which needs to be satisfied for a *Product*. Furthermore, an additional *impact* is used to score which violation accrues if this compliance is not satisfied. To be executed, each compliance is *implemented by* one or more functional *Test*.

Test represents the domain of a functional test, i.e., the code used for execution. It is used to define certain tests as general as possible and uses those in different *Compliances*. To yield certain *reusability* of such a *Test*, each one can use *variables* to create variability of specific properties, i.e., *test parameterization* is enabled.

The separation of code (*Test*) and demanded standard (*Compliance*) yields a technology's independence of the modeled *Compliance*. Thus, when using *Compliance* to demand *Product* standards, no or less knowledge on the concrete Test is required.

Later such elements are also often referred to as *Compliance Test*.

Audit outlines the final test execution. It *reviews* an *ApplicationSystem* to be compliant with the modeled *Compliance* derived through the operating *Product*. To ensure compliance, an *Audit generates and executes* specific *Tests* which are used to technically evaluate the formulated compliance, i.e., using code.

Report comprises the domain of reporting the results of a compliance test which was adapted, i.e., executed, on an *ApplicationSystem*. To this end, it needs to be aware of each *Compliance* and its respective results. Furthermore, the *Report* is used to *document* an taken *Audit*.

Based on this domain and its seven different *aspects*, we describe our *Continuous Compliance* process model in the following. In later sections, we refer to this general domain model and introduce a more detailed and technically motivated meta model for the *Continuous Compliance Testing* tool support (cf. Section 6.3) synthesizing this domain.

5.2. Continuous Compliance – Process model

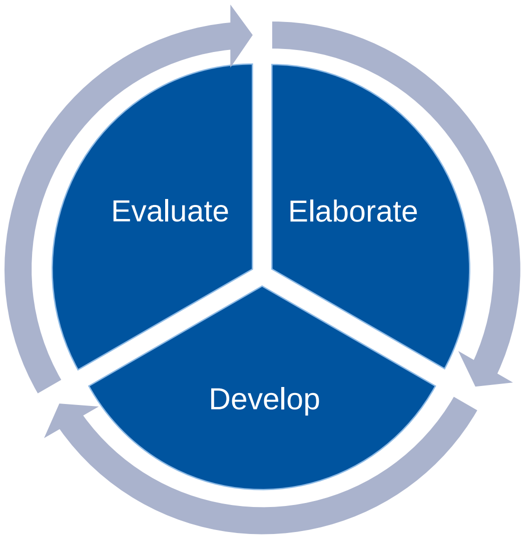


Figure 5.2.: An overall incremental and repeatable process model illustrating our proposed approach to solve the presented problems in a *continuous* manner.

Towards a *Continuous Compliance* process from a *product-point-of-view* as well as to tackle the central outlined problems, i.e., *System Misconfiguration*, *Information Management* and *Management Inexperience* (cf. Chapter 3), we propose a process model which is introduced in Figure 5.2. The main goal of this process is to formulate a guideline and activities on how to derive and verify product related compliance requirements.

We claim that this process, i.e., determining and crafting compliance tests from a *product-point-of-view*, needs to be realized with incremental and repeatable steps. This claim is justifiable since new compliance requirements can arise posterior or during a software development life cycle. Consequentially, we represented the process as a circular, self-repeating flow of phases to allow a *continuous* and incremental proceeding. It includes three central phases, i.e.: *Elaborate*, *Develop*, and *Evaluate*. Each phase has a set of activities which leads to a defined outcome required to succeed with the next phase.

We now explain each of those phases in more detail. To further strengthen our assertion of an incremental process, we additionally argue how each of these phases can be mapped to the *Software Development Process Life Cycle (SDLC)* (cf. Section 2.3). Afterward, in Section 5.2.2, we define and clarify each of the activities utilized to describe the phases and emphasize the mapping to an *SDLC* likewise. Next, we claim and define the required stakeholder and their *Roles* to conduct all activities. To conclude, we relate the presented responsibilities and roles in an superficial mapping which we present in Section 5.2.4.

5.2.1. Phases

To define the *Continuous Compliance* process model illustrated in Figure 5.2 more precisely, we briefly explain the purpose of each phase. Furthermore, to be able to perform each phase, we reference the required activities which are defined afterward. Finally, we discuss the mapping of each phase and its activities, i.e., the responsibilities to fulfill, towards *SDLC* stages.

Elaborate

The first phase, *Elaborate*, is applied to create the fundamental set of compliance requirements. Thus, the central assignment of this phase is to state and compile a set of all compliance requirements for the considered *Product* only and *not* the tested *ApplicationSystem*. That is, only product related compliance requirements independent from its operating infrastructure, i.e., *product-point-of-view*, should be created. This phase partially tackles the *System-Misconfiguration* and *Information Management* problems. On the one hand, concrete requirements, i.e., *Compliance*, are *elaborated* to ensure correct configuration, and on the other hand, such rules are defined and documented which reduces the problem of *Information Management*.

The *Compliance* requirements should be revealed as part of the *CCR-Analysis* activity and precisely defined as well as synthesized during the *CCR-Definition*. However, if similar compliance requirements occur which were part of the previous utilization of this process (or phase), reusing existing definitions is highly recommended. Reusability of *Compliance* definitions is legit, since the actual context, i.e., the final *ApplicationSystem*, is not yet considered or modeled (as demanded).

This phase and its activities, i.e., *CCR-Analysis* and *CCR-Definition*, can be mapped to the *Planning* and *Analysis* stages of an *SDLC*. During these stages, certain requirements (functional and *non-functional*) are defined. Thus, as we are interested in *non-functional* requirements as well (cf. Section 2.2), the *Elaborate* phase can be mapped to these stages.

Develop

The second phase, which we claim as part of the overall process model, is the *Develop* phase. Its central activity is to model the *Product* and all previous derived compliance policies (*CCR-Modeling*) on the one hand. On the other hand, this phase requires implementing (or reusing) the concrete, functional, *Tests* which are implemented to ensure the defined *Compliance* (*CCR-Implementation*). To this end, this phase aims to create the *Product* and *Test* aspects (cf. Section 5.1) of our model and relates those with the *Compliance* defined in the previous phase.

Due to accurate modeling, i.e., relating *Compliance* and *Product*, this phase reduces the problem of *System-Misconfiguration* since a demanded *Product* configuration is modeled. Furthermore, the issue of *Information Management* is reduced because relevant information from the *product-point-of-view* are modeled and thus documented.

The modeling (*CCR-Modeling*) activity of this phase can be mapped to the *Design* and *Implementation* stage of an *SDLC*. This claim is justifiable since, during the execution of these stages, certain product related design decision are taken which can be readopted to the *Product* modeling activity. The implementation of concrete *Tests* (*CCM-Implementation*) is similar to “normal” testing activities, and thus, mapping it to the *Testing and Integration* stage is a legitimate approach. Summarized, our *Develop* phase is covered by the *Design*, *Implementation*, and *Testing and Integration* stage.

Evaluate

Evaluate represents the last phase of our introduced process model. The prime task of this phase is to consider the results received by a compliance test execution (*Audit*), i.e., performing all derived *Test* modeled by a *Product*. First, the modeled *Product* needs to be *adapted*, i.e., test execution or simulation should be carried out (*CCR-Adaptation*). Afterward, the gained results need to be interpreted, and further consequences based on those results should be discussed (*CCR-Remediation*).

This phase comprises the *Audit* and *Report* aspects of the presented domain model and provides information on how compliant a *reviewed ApplicationSystem* is towards all involved stakeholder, e.g., the management. To this end, this phase primarily tackles the problem of *Management Inexperience*.

Our last phase can be mapped to the *Testing and Integration* and *Maintenance* stages of an applied *SDLC*. During the *Testing and Integration* stage, our *CCR-Adaptation* activity would take place and our remediation would be part of the further *Maintenance*.

In accordance with *CCR-Remediation*, a next process iteration would be initiated, e.g., if the modeled compliance information seems to be insufficient, or the considered *ApplicationSystem* needs to be fixed regarding its failing compliance requirements.

To conclude, the presented phases of the *Continuous Compliance* process model are capable of reducing and partially solving the outlined (central) problems. Additionally, each phase can be mapped to the very fundamental *SDLC* process model, which strengthens the assertion of an incremental and iterative process. Additionally, it allows a more concrete responsibility and role definition on which we elaborate in the next sections.

5.2.2. Responsibilities

For a well-defined utilization of the proposed *Continuous Compliance* process model, we claim six different *Continuous Compliance Responsibilities (CCR)* categories, i.e.: *CCR-Analysis*, *CCR-Definition*, *CCR-Modeling*, *CCR-Implementation*, *CCR-Adaptation*, and *CCR-Remediation*. Their naming indicates that these categories are similar to existing activities in the SDLC (cf. Section 2.3). Hence, the relation to an applied SDLC-based process model is more comprehensive, and an integration is justifiable. The numbering is used for reference as part of the roles and responsibilities mapping in Section 5.2.4.

CCR-Analysis groups all activities related to the fundamental analysis of the considered *Product*. It needs to be defined `(1)`, and further general, vendor specific, compliance policies should be added `(2)`. Moreover, the importance (*impact*) of all related software entities needs to be defined `(3)` to enable a meaningful violation score calculation and reporting. Finally, the violation remediation plan needs to be outlined to handle test fails accordingly `(4)`.

CCR-Definition summarizes the task of formulating and defining *Compliance* requirements. It should precisely describe what policy is required `(5)` and which violation impact this characteristic on system misconfiguration or security has `(6)`.

CCR-Modeling comprises all modeling related actions. The considered *Product* should be described and modeled `(7)`. Thereby, a possible ruse of already defined *Products* and its composition should to be considered (for reuse) `(8)`. Furthermore, when reusing *Products* and therewith its *Tests*, the *Test* context needs to be (re-) defined to terminate the test genericity (cf. *Test Genericity* Section 6.4.1), i.e., a redefinition of placeholder values `(9)`.

CCR-Implementation focuses on the responsibilities and tasks on preparing executable *Tests* to ensure certain *Compliance*. First and foremost, all required test functionality needs to be implemented `(10)` and equipped with placeholders to enable test genericity (cf. *Test Genericity* Section 6.4.1) `(11)`. Those tests than need to be linked to existing compliance policies to facilitate their adaptation `(12)`.

CCR-Adaptation comprises of the competencies of the final compliance-test adaptation. Primarily, the test execution has to be planned and carried out `(13)`. However, to be able to check a hosting system for compliance, the related infrastructure needs to be defined `(14)` and linked to the considered *Product*, i.e., the *ApplicationSystem (SUT)* is modeled.

CCR-Remediation describes certain activities which face the result evaluation and their rating. First and foremost, the reporting of one test execution needs to be conducted. Based on these result, i.e., the maximum compliance violation, the compliance index and detailed test results (cf. Section 6.4.3), further actions have to be considered `(15)`. Besides, a possible infrastructure remediation should be taken `(16)`.

5.2.3. Roles

In the following, we claim a set of involved roles to derive product related compliance requirements successfully, i.e., those roles and stakeholders which can yield the previous (indirectly) noted information on the *Product* and its *Compliance*.

The resulting set represents a narrowed superset of various roles – involved into the field of product security – defined in an incremental and iterative software development life cycle like *SDLC* or *SDL* (cf. Section 2.3). Even more, we evaluated public standards, such as ITIL or ISO 27001, and CMMs facing software security, such as BSIMM or

OpenSAMM. For example, the OpenSAMM claims that the roles *Manager*, *Business Owner*, *Architects*, *Developers*, *Security Auditor* and *QA Tester* [OWA17a] are part of the overall process to reach a certain security level. Thus, to allow a simple integration in existing processes and an assistance of security standards (cf. Evaluation in Chapter 8), we specified those roles.

Next, we briefly introduce each synthesized role, mention its tasks during *Continuous Compliance* and indicate possible substituting roles, i.e., if the described role is not (yet) adopted.

Chief Security Officer (CSO) covers the physical security of a company. The central assignment is to protect people, assets, infrastructure and the technology. The CSO takes over the leadership required to identifying, assessing and prioritizing violations. In large scale organizations, the CSO often works in close collaboration with the *Chief Information Security Officer (CISO)* (cf. [Tec17]).

In the process of determining compliance requirements for software products, the roles assignments are to outline and describe the vendor's overall compliance. Thus, the CSO contributes to the fundamental *Compliance* every *Product* has to adhere to. Further specific requirements, which are not familiar to the development team, should be documented.

If the CSO is not yet established its responsibilities should be passed to the *Product Owner*.

Product Owner (PO) is a development role as part of the incremental and iterative process *Scrum*. This person is responsible for working with different, participating, groups to determine which features should be implemented (cf. [Tec17]).

In the process of determining compliance requirements for software products, the roles assignments are to define the compliance policies implied by each product feature. Moreover, a PO should be responsible for the organizational processes, i.e., collecting all other relevant compliance requirements from the SA, SD or QA for its specific product.

If the PO is not yet established its responsibilities should be passed to the next higher management level, i.e., the person who accounts for the product/software.

Software Architect (SA) is an expert role in the overall software development team. The roles main task is to establish the high-level design decision for the system architecture and to decide technical standards as well as fundamental concepts regarding the software components interactions. The perspective of an SA is more abstract than those of an SD.

In the process of determining compliance requirements for software products, the roles assignments are to define and to model the software architecture with the help of our proposed *Compliance Management Tooling*, e.g., software components should be created and combined as described in the architecture. Additionally, the SA should add further policies which are applied to certain architecture or communication constraints.

If the SA is not yet established its responsibilities should be passed to the development team (SD) as they are accounting for the software architecture in this case.

Software Developer (SD) describes those persons who are involved in the concrete realization of a software component or artifact, i.e., designing, programming, and testing. To this end, a software developer has very detailed knowledge about the software/system's functionality and its dependencies. The roles detailed tasks could vary in terms of the position in the overall development team.

In the process of determining compliance requirements for software products, the roles assignment is to add specific policies which are implied throughout the implementation, e.g., the availability of special third-party libraries or tools.

The role of the SD is not replaceable since each software has at least one developer. However, its duties could be broadened if certain other roles, e.g., SA or PO are not available.

Qualitiy Assurance (QA) is a group of persons who undertake administrative and procedural activities to prevent failures or defects in the resulting software. Thus, they are responsible for the overall software quality. Usually, this role is associated with the software tester which has to define and execute certain tests to check and verify the correctness of a software artifact.

In the process of determining compliance requirements for software products, the roles assignment is to outline software quality properties which could have an impact on the security. For example, non-functional constraints, such as responsiveness, should be expressed if they are necessary to guarantee certain functionality.

If the QA is not yet established its responsibilities should be passed to the development team (SD) as they are accounting for the software quality in this case.

Build / Release Manager (BM/RM) supports the software development team in terms of managing changes to the IT infrastructure, such that these changes are performed in a secure, documented, and efficient manner. Moreover, planning and observing of (final) software roll-outs are parts of the roles responsibilities (cf. [IT]).

In the process of determining compliance requirements for software products, the roles assignment is to compile all the needs the software requires from on the hosting infrastructure, e.g., driver or network protocols, and *vice-versa*. Besides infrastructure related policies, additional policies required through roll-out and delivery process demands, e.g., configuration parameters, should be added as well.

If the BM/RM is not yet established its responsibilities should be passed to the SA since its knowledge on the infrastructure is most detailed in the project.

5.2.4. Roles – Responsibilities Mapping

Finally, in Table 5.1, we present our recommended mapping between the defined roles and responsibilities. Since all single parts are previously outlined in detail, we refrain from presenting a detailed argumentation on this mapping. Instead, this mapping serves as a short and accurate summarizing of the previously described roles and its responsibilities.

To summarize, all presented roles and responsibilities can be mapped to each other. However, this mapping can vary if certain roles are not yet deployed in a company (cf. substituting roles in Section 5.2.3).

5.3. Summary

In this chapter, we have presented our broad concept for an incremental process, i.e., *Continuous Compliance*, and outlined its assistance to tackle the introduced problems.

First, in Section 5.1, we introduced and defined the overall process domain. We explained the general concepts of *Product*, *Compliance*, and *Test*. Furthermore, we described the adaptation of the *separation of concerns* principle, i.e., additional modeling layer (*Product*) and the option to reusability, and declared the *ApplicationSystem*, i.e., the resulting context for each model.

In Section 5.2, we presented our general process model to derive, create and test compliance requirements from a *product-point-of-view*. To specify each phase in more detail, we discussed and claimed roles and responsibilities which are required to successfully derive compliance requirements for an SUT. It is to conclude that, *Continuous Compliance* from a *product-point-of-view* is a complex (and reoccurring) assignment, including different departments and individuals. Nevertheless, with the support of the provided roles and their responsibilities, the accomplishment of this process should be clearer and more straightforward. To simplify its adoption, we have presented an exemplary mapping to the *Software Development Life cycle (SDLC)* which emphasizes the applicability of *Continuous Compliance*. Further integration and support are shown as part of our Evaluation (cf. Chapter 8).

To promote the practical application of this process called "*Continuous Compliance Testing*", we present a more technical motivated concept, i.e., a tool support for our process and its specific modeling domain (*separation of concerns*, cf. Section 5.1), in the following.

Responsibilities	Chief Security Officer	Product Owner	Software Architect	Software Developer	Quality Assurance	Release Manager
CCR-Analysis						
`(1)` ... *defines customer-project*	-	✓	-	-	-	-
`(2)` ... *defines general, vendor specific, policies*	✓	✓	-	-	✓	✓
`(3)` ... *defines violation impact of involved artifacts*	✓	✓	-	-	✓	-
`(4)` ... *describes violation remediation*	✓	-	-	-	✓	✓
CCR-Definition						
`(5)` ... *precisely describes policies*	✓	✓	✓	✓	✓	✓
`(6)` ... *defines violation impact of compliance*	✓	✓	-	-	✓	-
CCR-Modeling						
`(7)` ... *models Product*	-	-	✓	✓	-	-
`(8)` ... *considers reuse of already modeled Products*	-	✓	✓	-	-	-
`(9)` ... *placeholder (re-)definition for artifact*	-	✓	✓	✓	-	-
CCR-Implementation						
`(10)` .. *implements functional tests*	-	-	-	✓	✓	✓
`(11)` .. *improves genericity by use placeholder*	-	-	-	✓	✓	✓
`(12)` .. *links tests to compliance entities*	-	-	-	-	✓	✓
CCR-Adaptation						
`(13)` .. *plans and execute Tests*	✓	-	-	-	✓	-
`(14)` .. *setups environments for customer-projects*	-	-	✓	-	-	✓
CCR-Remediation						
`(15)` .. *considers test results and plan further actions*	✓	✓	-	-	-	-
`(16)` .. *suitable infrastructure remediations*	-	-	-	-	✓	✓

Roles: Chief Security Officer, Product Owner, Software Architect, Software Developer, Quality Assurance, Release Manager

Table 5.1.: Roles and Responsibility Mapping – A checkmark (✓) indicates that this task is part of the responsibilities of the linked role. A single task can be delegated or distributed to multiple roles, which corresponds to more than one check mark in a row. A dash (-) indicates the opposite, e.g. due to missing knowledge or range of duty, that the role should not be mapped to this task

6. Continuous Compliance Testing

There are no facts,
only interpretations.

FRIEDRICH NIETZSCHE

Contents

To yield a practical adaptation of *Continuous Compliance*, we present a more technical concept leading to a tool support and briefly describe how to support each phase. The resulting procedure, i.e., the tool-supported application, is called *Continuous Compliance Testing*, where "testing" indicates the active adaptation of *Continuous Compliance*.

The proposed tooling, *Compliance Management Tooling (CMT)*, should support all involved individuals (roles, later often referred to as *user*) during each phase. Thus, it needs to be likewise capable of *defining*, *modeling*, and *executing* compliance requirements. In Section 6.1, we briefly outline the tool support for every phase to highlight the mapping between both concepts, i.e., the general *Continuous Compliance* process model and its tool-supported application (*Continuous Compliance Testing*). The concrete needs for this tooling are explained using natural language in Section 6.2, and afterward, in Section 6.3, we use those requirements to define the fundamental (more technical) meta model.

6.1. Continuous Compliance Phase Support

To accomplish a tool support for the previously presented *Continuous Compliance* process model, an accurate mapping and realization of each process phase is required. To this end, we extended the general model with additional, more precise, activities. These extensions are indicated with a small, named subcircle in Figure 6.1. We describe how each of those activity helps to perform the more general phase, e.g., how the *Defining* activity supports the *Elaborate* phase, in the following.

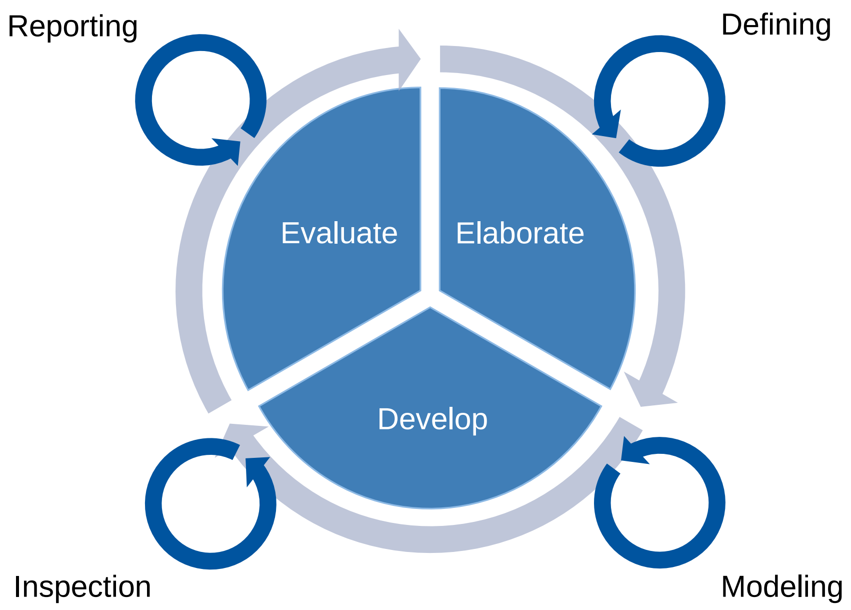


Figure 6.1.: The *Continuous Compliance* process model extended with sub-activities highlighting the tool-supported fields.

Elaborate During the *Elaborate* phase, a primary tooling needs to support the user in *defining* compliance requirements. For example, it needs to provide functionality to create and manage various compliance requirements. Additionally, it should support the reusability of previously created compliance tests. To this end, the single and briefly described activity during this phase is the *Definition*.

Develop During this phase, a tool should support the modeling and implementation of *Tests*, indicated by the second subcircle (*Modeling*, cf. Figure 6.1). In particular, it should support the user in modeling the relations between the central aspects of our modeling domain. For example, adding certain *Compliance* requirements to an existing *Product* should be supported. To enable an efficient modeling, the tool should provide a proper support for reusability, i.e., reusing already existing elements in other (maybe varying) contexts[1]. Moreover, a tooling should encourage the functional implementation of *Tests* to ensure the *Compliance* validity. Thus, creating and adding *Tests* to already modeled *Compliance* requirements is a necessity.

To provide the essential input for the next phase, i.e., results of an compliance test, a proper tooling should provide an ability to execute the required tests which is indicated by the third activity, i.e., *Inspection*, in the process model.

[1]At this point, *context* does not reference the *ApplicationSystem* but rather another *Product* or *Compliance* reusing already existing *Compliance* or *Tests*.

Evaluate To assist the evaluation and its related decision making, a *Reporting* functionality is somehow needed. Thus, a tooling supporting the *Continuous Compliance* process should have the functionality to present the test results, i.e., which of the defined *Compliance* requirements passed or failed. Furthermore, it should report the resulting violation score calculated on the basis of the previously modeled *impacts*. In more detail, we claim that an appropriate tooling should support the reporting classes of *Quality assurance*, i.e., trying to eliminate security vulnerabilities, and *Technical Oversight*, i.e., security status or posture of an IT system, as defined by [Jan10; Sav07; VHS03].

To conclude, a tooling should support the decision and remediation process with data condensed by the existing test results.

Based on this rough overview on how to support the process more technically, we describe additional and more fine grained requirements for the *Compliance Management Tooling* in the next section.

6.2. Software Requirements Specification

To provide a tool support which tackles the lack of non-existing software for modeling compliance requirements from a *product-point-of-view* (cf. Section 3), we propose the *Compliance Management Tooling (CMT)*. To enable a practical application of the previous introduced *Continuous Compliance* process model, the tooling has to support each phase with various functionalities (cf. Section 6.1). We describe all requirements in natural language [Poh10] indicate certain requirements with an marker, i.e., *[CMT-Req-X]* where X is ongoing numeration, to support an easier cross-referencing, e.g., to assist the verification as part of our evaluation (cf. Section 8.1).

Using the CMT, a user should be able to model *Compliance* for a *Product* to gain the ability to ensure compliance from a *product-point-of-view*. To model a *Product* in more detail, i.e., more fine-grained elements to reinforce *Reusability* of partial *Products*, the user should be able to represent it through *Artifacts*, such as *Software Components* and *Software Landscapes* [**CMR-Req-1**]. For example, a LAMP web server could be modeled as a *Software Landscape* using a Linux, Apache, MySQL, and PHP *Software Component*. This segmentation yields higher reusability since one *Software Component* can be used in other *Products*, e.g., PHP as part of another WAMP[2] server, and decreases the problem of *Information Management* as existing information can be reused. Likewise, these more fine-grained *Artifacts* can *demand* certain *Compliance*.

To unfold the required *Compliance* domain (cf. Section 5.1), the tool should allow linking this compliance against existing *Artifact* entities [**CMR-Req-2**]. Furthermore, it should provide more fine grained entities to describe the *Compliance* at all and similar to the *Artifact* entities reduce the problem of *Information Management*.

In particular, the tooling should supply *Compliance Profiles*, *Compliance Rule Sets* and *Compliance Rules* to support an excellent modular construction of certain *Compliance*

[2]Windows, Apache, MySQL, PHP

[**CMR-Req-3**]. In general, *Compliance Rules* should be groupable as *Compliance Rule Sets* and further as *Compliance Profile*. This aggregation should allow the usage of *atomic Compliance Rules*, i.e., exactly enforcing one requirement. For example, the previously mentioned PHP *Software Component* demands a *running* `php-fpm` service which needs to be *installed* as well. To achieve the best *Traceability*, i.e., being able to spot the failing compliance easily, this dependency has to be modeled with two distinct and atomic *Compliance Rules*, each enforcing one requirement (installed *or* running). To finally model the specific service requirements (installed *and* running) in a more simple way, both atomic *Compliance Rules* can be aggregated into one *Compliance Rule Sets*.

To ensure that each demanded *Compliance* is satisfied by the *ApplicationSystem*, the tooling should provide the ability to define and link functional *Tests* [**CMR-Req-4**]. This feature finally resolves the problem of *System-Misconfiguration* as a user has now the ability to model and to *check* a system for compliance.

Each *Test* should encapsulate certain security tests and evaluation functions as code to *test* its *implementing Compliance*. Further, it should allow using variables [**CMR-Req-5**], e.g., a variable (*Placeholder*) for an expected result of an evaluation function, which could be defined on a higher level. That is, in terms of to the previous presented domain model, the level of *Compliance* and *Product*. Exemplary, this feature should allow the user to verify a certain value which is defined in the concrete *Product* context, e.g., an existing file. Thus, the user is able to define *Tests* in a general manner and can reuse them in different contexts.

Finally, beside modeling the compliance requirements for a certain product using *Tests*, *Compliance* and *Artifacts*, the user should be able to create the concrete *ApplicationSystem*. Therefore, the CMT needs to provide a *Customer Project* entity, which allows the combination of a *Product* and *Environment* [**CMR-Req-6**]. Using this entity the user should be able to execute or simulate the compliance test [**CMR-Req-7**]. Besides the self-explaining execution, the model simulation should be considered to *simulate* a compliance test and to check for certain properties, e.g., if at least one modeled *Tests* per *Artifact* exists or if all used *Placeholder* are defined through linked *Placeholder Values*.

To determine the violation score impact if such a *Compliance* fails, the user should have the ability to define an *impact* on *Compliance* and *Artifact* level [**CMR-Req-8**]. The accumulation of this impact values should yield the final violation score for a particular test. Then, the CMT should be able to report on those compliance violation score and further test results, e.g., which *Compliance* failed or passed. More precisely, a *Reporting* functionality is necessary [**CMR-Req-9**] to support the *Management Inexperience*

6.3. Meta-Model

Models are powerful tools to express the structure, behavior, and other properties [Spr+10]. To yield such a *powerfull tool* and to express the permitted structure to which all following models must adhere to [Spr+10], we us the well-know *Unified-Modeling-Language* [Boo05] to craft our modeling language. To be able to model all previously defined aspects, the meta model is based on the claimed requirements for the *Configuration Management Tool (CMT)* in Section 6.1 and Section 6.2.

By its origin this meta model relies on the domain model of the *Continuous Compliance* process model (cf.Section 6.3) which was introduced in the previous chapter. Thus the overall meta model can be generalized – for the sake of simplicity of further characterizations – in four major areas illustrated in Figure 6.2.

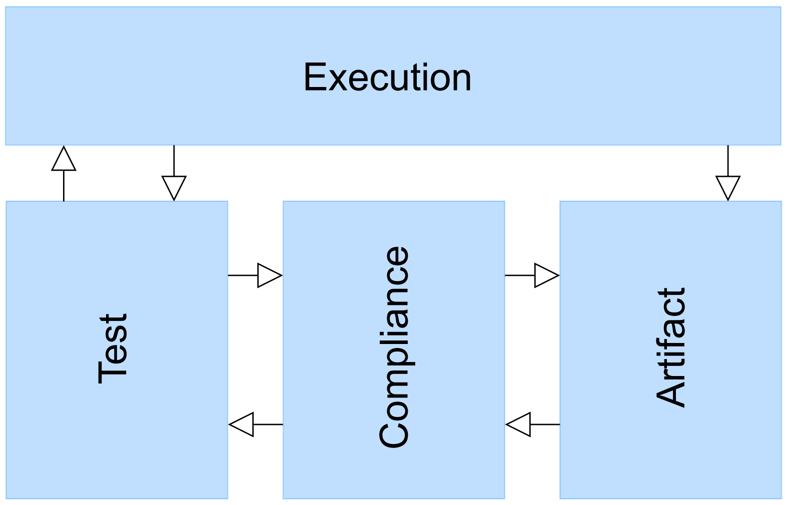


Figure 6.2.: Generalized Compliance Management Tooling Meta Model containing the four major areas and its interconnection

As shown in Figure 6.2, the central meta model aspects, namely *Test*, *Compliance* and *Artifact*[3] reoccur and are horizontally interconnected between themselves which indicates the similarity (and usage of the outlined domain model). Furthermore, *Test* and *Artifact* are vertically connected to a (new) *Execution* domain, which contains the aspects of compliance test: *execution* and *simulation*. Each single aspect, its specific modeling elements, and interconnection to other aspects are explained in detail in the upcoming paragraphs. The resulting and fully composed meta model is summarized at end of this section (cf. Figure 6.7).

[3]Previously referred to as *Product*

CMT Meta Model – Test

The most crucial part of the domain model is represented by the *Test* field. It contains the whole testing domain and all its required information to model and finally generate one concrete, executable compliance test.

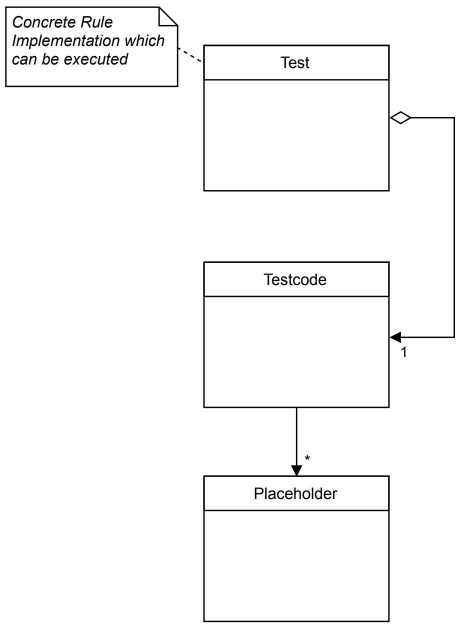


Figure 6.3.: *Test* aspect of the *CMT meta model.* To introduce the variables in *Testcode* the aspect *Placeholder* is presented

The prime element, which is used by the *Compliance* and *Execution* domain, is the *Test* entity. It serves as a single representative for a test and depends on a *Testcode* entity used to formulate the specific test code. To describe a *Testcode* as general as possible, it could include as many *Placeholders* as needed. For example, a port test[4] could be more generalized by replacing the concrete port number with a placeholder.

CMT Meta Model – Compliance

The *Compliance* domain encapsulates the information (impact, significance and classification, cf. attributes of Figure 6.4) required for violation score calculation and reporting. Moreover, it acts as intermediate domain and establishes the link between the *Tests* and *Artifacts.* The domain includes three different manifestations of the compliance entity, namely *Compliance Profile*, *Compliance Rule Set*, and a *Compliance Rule.* This split enables a high degree of reusability and various level of granularity. Furthermore, compliance information can be aggregated and shared between these different manifestations through a hierarchical-like structure. With this *Generalization Hierarchy*, a result of the

[4]For example, testing whether port 80 is opened to serve standard HTTP requests.

hierarchical decomposition design principle, the overall complexity of this data structure is reduced.

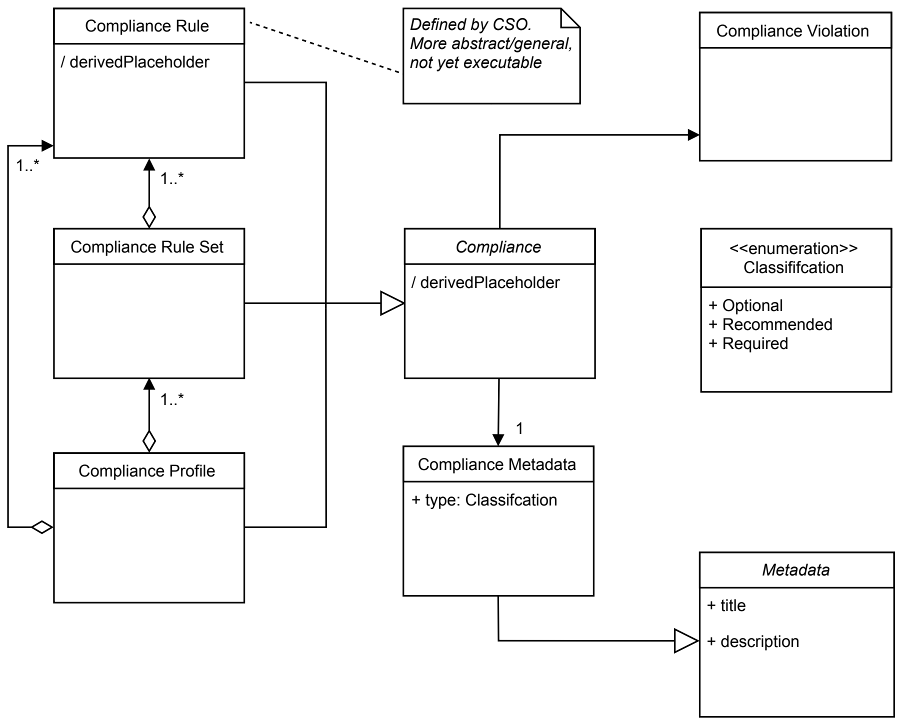


Figure 6.4.: *Compliance* aspect of the *CMT meta model*. It illustrates the composition of *Compliance*. Additionally, the used *Metadata* and *Compliance Violation* are shown.

For example, a basic *Compliance Rule*, e.g., testing correct port configuration for a network service, could be used in multiple *Compliance Rule Sets*, e.g., Apache and Nginx services. These *Compliance Rule Sets* can likewise be aggregated into one (or more) *Compliance Profile(s)*, e.g., a web service using an Apache and FTP service.

This domain area is connected to all three other domain areas and represents one essential field as well. The *Test* domain area is associated throughout the *Compliance Rule* entity, since every rule describing a certain compliance requirement needs some or at least one test to accomplish the evaluation. Since these *Compliance*, modeled with the help of this domain area, are used to described *Artifact* compliance policies, each *Artifact* can have as many *Compliances* as needed.

CMT Meta Model – Artifact

The primary application of the *Artifact* domain is to define the context for each derived *Test* (through the intermediate *Compliance* domain) and to model the software and its components for which the compliance requirements should be defined. Though each *Artifact* is linked to various compliance entities (*Compliance Rule*, *Compliance Rule Set* or *Compliance Profile*), which describe the demanded compliance requirements. This inter-domain connection allows a later derivation of compliance requirements based on the input a single software artifact.

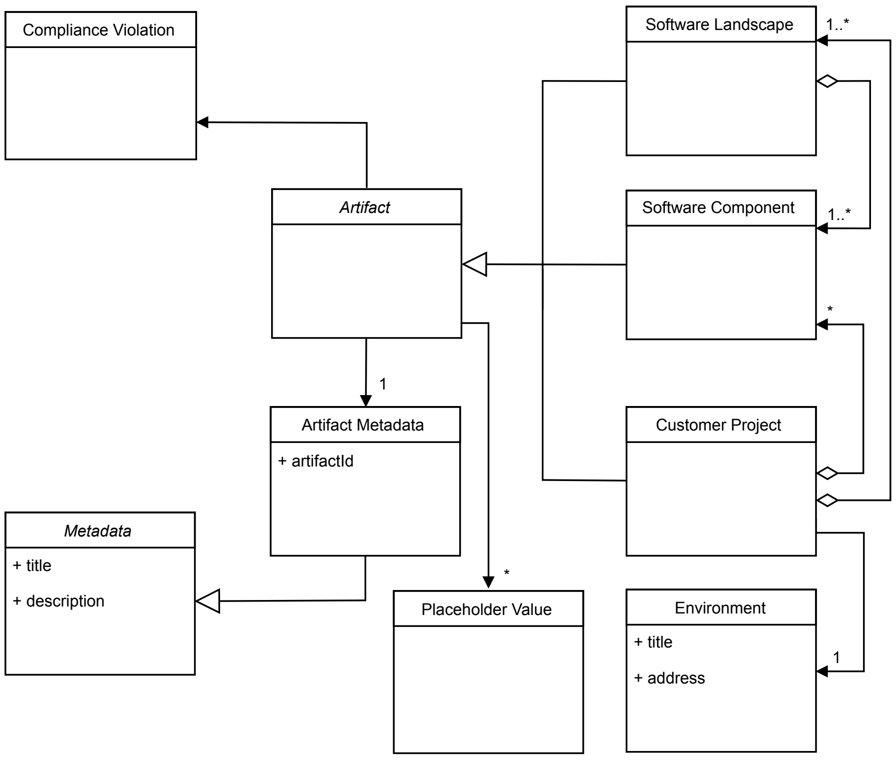


Figure 6.5.: *Artifact* aspect of the *CMT meta model*. It illustrates the composition of *Artifact* and the used *Metadata* as well as the *Compliance Violation* are shown. Additionally, to be able to define a value for derived *Placeholder* the *Placeholder Value* aspect is presented.

Similar to the *Compliance* domain, the *Artifact* domain introduces a hierarchical-like structure to reduce the complexity. Three manifestations of *Artifact* exist as well and each is associated to one or more *Compliance* entities to establish a software-compliance relation. *Software Components*, *Software Landscapes*, and *Customer Project* are introduced to represent the software and its underlying structure. For example, a *Software Landscape* can have and reuse arbitrary many *Software Components*. To this end, a compliance requirement needs to be modeled only once and can be reused as often as required.

Depending on the traced modeling approach, a single *Software Component* can be interpreted as a system or software component which owns a process on the hosting system. Accordingly, a *Software Landscape* is an accumulation of various “stand-alone” processes which are used to accomplish the desired functionality. For example, a ssh-service could act as a *Software Component* as part of a web server *Software Landscape*.

As the *Artifact* precisely defines the context for the (indirectly) linked *Tests*, allowing the definition for the previously – in the *Test* aspect – introduced *Placeholder* is necessary. This necessity is realized through an entity *Placeholder Value*, which itself is linked to one *Placeholder Value* entity from the *Test* domain. To counter an ambiguous definition of placeholder values, which can arise through hierarchical inheritance, the latest artifact defining the *Placeholder Value* overrides all previous values. This central idea (*Reusability*) and the derivation, as well as the overwriting process, are described detailed in Section 6.4.1.

CMT Meta Model – Execution

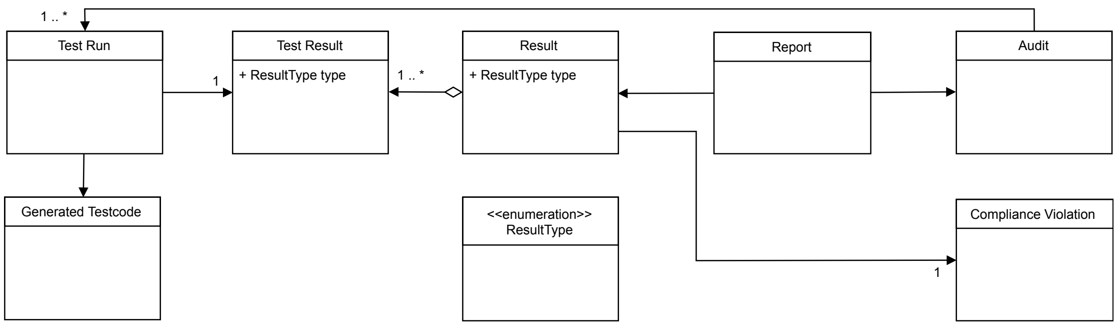


Figure 6.6.: *Execution* aspect of the *CMT meta model.* It illustrates the composition of an *Audit.* Additionally, aspects to store and report the tests results (*Test Run*, *Test Result*) are shown.

The *Execution* domain model encapsulates all required information which are needed during and after a compliance test run. *Artifacts*, *Compliance Rules* and *Tests* are snapshotted and represented through the Entities *Audit*, *Result* and *Test Result* accordingly.

This domain holds all kinds of information and thus relies on all three, previously introduced, domains *Test*, *Compliance* and *Artifact.* However, this additional domain area is required and should be separated due to the conception of auditable and repeatable compliance reports. Those reports are possible as the “status-quo” is stored (snapshotted) during a test and can be used to repeated the test-run under the same conditions. *Generated Tests*, which rely on *Test*, *Compliance* and *Artifact* domain information, are stored and linked by a single *Audit.* Besides the generated tests, further information on the environment-under-test needed to be persisted as well, which is achieved by an *Audit* entity as well.

CMT Meta Model

The entire meta model which describes the central aspects of our proposed *Compliance Management Tool* is summarized in Figure 6.7. It shows all four, previously explained parts, i.e. *Test*, *Compliance*, *Artifact* and *Execution*, and their connection between themselves.

In further section of this thesis, a *model* is an full or partial instantiation of aspects from the presented *meta-model* illustrated in Figure 6.7. This implies that we make use of the ability to specify and adopt selective visualizations (also known as aspects or viewpoints) of our meta-model [Spr+10], i.e., we omit certain parts of the modeling language like the *Execution* part.

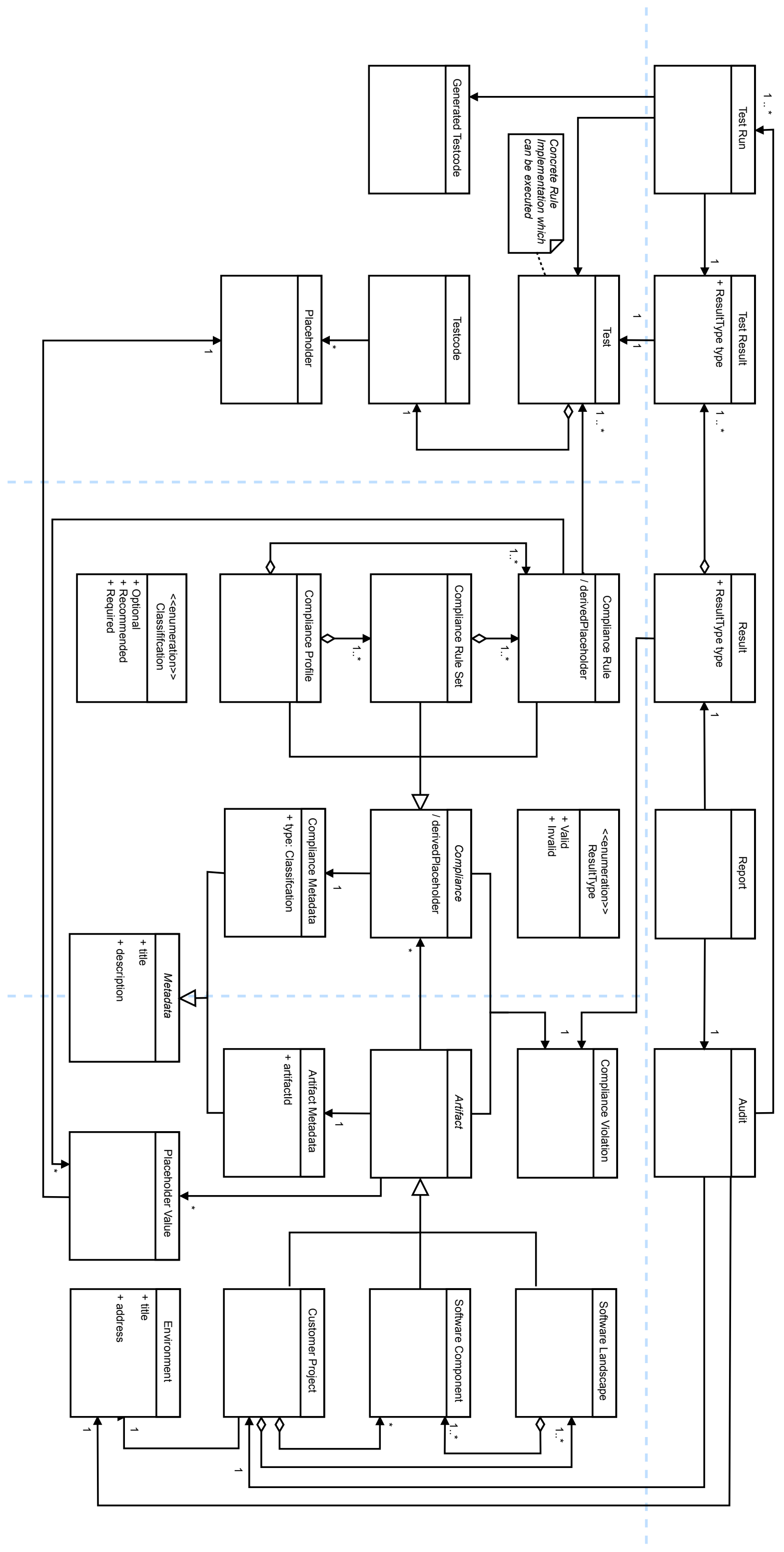


Figure 6.7.: Entire CMT meta model to precisely define the modeling domain. The blue swim-lanes indicate the previously defined aspects, i.e., *Test*, *Compliance*, *Artifact* and *Execution*

6.4. Technical Concepts

In this last section, we define further, more technical oriented, concepts to match the *Compliance Management Tooling* as supporting tool for the *Continuous Compliance Testing* process. To this end, we rely on the previously outlined software meta model (cf. Section 6.3) and describe functionalities based on its aspects.

First and foremost, we present a concept to increase the reusability of already created *Tests*. Next, we briefly describe the *Execution* of compliance test in Section 6.4.2 and continue with the concept of *Reporting*.

6.4.1. Reusability

Due to the problem of *Information Management*, i.e., handling a large amount of similar information, we introduce the concept of *Reusability*. Given our proposed software meta model, certain reusability is given implicitly. For example, one can create and model a *Compliance Rule* once and reuse it – due to a separation of concerns (cf. Section 5.1 and Section 6.2) – in other *Product* scopes simultaneously. However, as certain concrete compliance properties, such as *filename*, *port*, *settings*, or *locations*, can change across different *ApplicationSystems*, one needs the option to somehow “redefine” certain attributes in the according to test code. The simplest solution is to create a specific test for each changing property. However, this approach leads to an enormous test code redundancy. To this end, we mentioned the usage of *Placeholder* (variables) to allow test code parameterization (cf. Section 6.2). We present three minor concepts *Test Genericity*, *Whitelisting*, and *Blocking* to precisely clarify the application and cancellation of variables in test code.

Test Genericity

To accomplish a high reusability of already implemented *Compliance Test* we introduce our concept *Test Genericity*. We describe its *Basic Idea* and the *Concrete Concept* in the following.

Basic Idea The basic idea of *Test Genericity* is to allow custom, user-defined and overwriteable values in the compliance test code. These *Placeholders* could be compared to regular variables used in common programming language. However, to increase the flexibility even more, *Placeholder* can be instantiated with an expression to define a range or a set of certain values.

A *context* for one or more *Placeholders* is defined through the superiorly linked *Artifact* entities (cf. Section 6.3). The instantiation, i.e., deriving the actual values for a given *Placeholder*, should be performed during the test generation process.

Concrete Concept To yield *Test Genericity*, a test can have arbitrary many *Placeholder* entities linked. Each *Placeholder* is then defined with the help of a *PlaceholderValue*, which belongs to an *Artifact* (cf. *Test* and *Artifact* meta model in Section 6.3). In addition, defining a very general test and instantiating it (defining its variables) later is possible, e.g., when the (runtime) context is known.

A *Placeholder Value* is represented through a *Compliance Management Tool – Expression (CMT-Exp)*. The *CMT-Exp* is described using a *Context Free Grammer (CFG)* (Equation Set 1). As defined, the *CMT-Exp* allows recursion at certain points, which allows fine-grained value description. Creating *Ranges* of values (`..`), creating *Sets* of values (`[]`), and using value *Exclusion* (`!`) is supported. For example, the expression `[a, b, 0 .. 9, !3 ]` is evaluated to the set `{a,b,0,1,2,4,5,6,7,8,9}` and is similar to the (more complex) set definition `[a, b, 0, 1, 2, 4, 5, 6, 7, 8, 9]`. Applying this example *Placeholder Value* on a *Placeholder* defined for a test would then result in 11 unique tests instances.

$$Symbols = \text{Set of all alphabetic and numeric characters}$$

$$\begin{aligned}
G &= (\{Exp, Set, SetItem, Value, Character\}, Symbols, R, Exp)\\
R &= \{\\
Exp &\rightarrow \quad Value \quad | \quad Set\\
Set &\rightarrow \quad [\quad SetItem \quad]\\
SetItem &\rightarrow \quad Item, SetItem \quad | \quad !Item, SetItem \quad | \quad \epsilon\\
Item &\rightarrow \quad Value \quad | \quad Value..Value\\
Value &\rightarrow \quad CharacterValue\\
&\}
\end{aligned}$$

$$\begin{aligned}
&Character \in Symbols\\
&\text{G over the alphabet} \quad A = \{\ [\ ]\ ,\ .\ !\ \epsilon\ \}
\end{aligned}$$

Equation Set 1: A Context Free Grammar (CFG) representing the expression (*CMT-Exp*) used to define placeholder values.

This concept supports to rely on one test, e.g., a test checking for open ports, in multiple, different contexts, e.g., web-server on port 80 and ssh-server on port 22. Furthermore, the ability to override *Placeholder Values* is created. In the next section, we elaborate on how overwriting should be handled and how overwriting creates the foundation for a whitelisting of tests.

Context	**PlaceholderValue** as *CMT-Exp*
A	`[ 8080, 8081 ]`
B	`[ 0 .. 1024, !22, !443 ]`
C	`[ 0 .. 1024, !22 ]`

Table 6.1.: Compiled set of all three contexts A, B, and C, for the *Test* CT3. For each context the derived *PlaceholderValue* expression is given.

Whitelisting

To extend the practicalness of *Test Genericity*, we present a concept on *Placeholder* overwriting, called *Whitelisting*. We describe its *Basic Idea* and the *Concrete Concept* in the following.

Basic Idea *Whitelisting* allows the definition and modeling of new *CustomerProjects* more simplified. In a common scenario, all tests for one, unique *CustomerProject* are properly defined for this explicit case with no reusability in mind, e.g., a J-Unit test for a specific class. Furthermore, all vendor related policies, i.e., those independent from the specific *CustomerProject* and applicable, need to be readopted each time.

To simplify and speed up the process of defining a new *CustomerProject*, we present the concept of Whitelisting. The idea is to create a set of *ComplianceTests* (*Baseline Tests*) as strictly as possible once, i.e., blacklisting as much as possible. Next, these *Baseline Tests* should be part of any *CustomerProject*. To be applicable in any context, the support for whitelisting is required, i.e., relaxing certain values. Thus, *Whitelisting* denotes the process of *PlaceholderValue* relaxing through *CMT-Exp* (cf. Section 6.4.1).

Concrete Concept The concrete concept on *Whitelisting* is defined and illustrated through a fictional test scenario which is partially modeled in Figure 6.8 using the previously defined meta model (cf. Section 6.3). The figure shows the three central aspects, i.e., *Artifact*, *Compliance* and *Test*, illustrated as different layers. It represent the full derivation tree of tests for one *CustomerProject* P. Each layer represents another partial domain (cf. Section 6.3), starting with *Artifacts* on top (Layer 1), followed by *Compliances* (Layer 2) in the middle and finished with the *Test* domain at the bottom (Layer 3). The highlighted paths represent all possible derivation paths from *CustomerProject* P to a *Test* CT3, which models a port test in this scenario.

Applying *Whitelisting* in this scenario will result in three disjoint contexts (highlighted paths) for the inherited *ComplianceTest* CT3, context `A` (leftmost), `B` (center) and `C` (rightmost). The derived *Placeholder Values* are complied in Table 6.1. Their derivation is explained in detail in the following three steps.

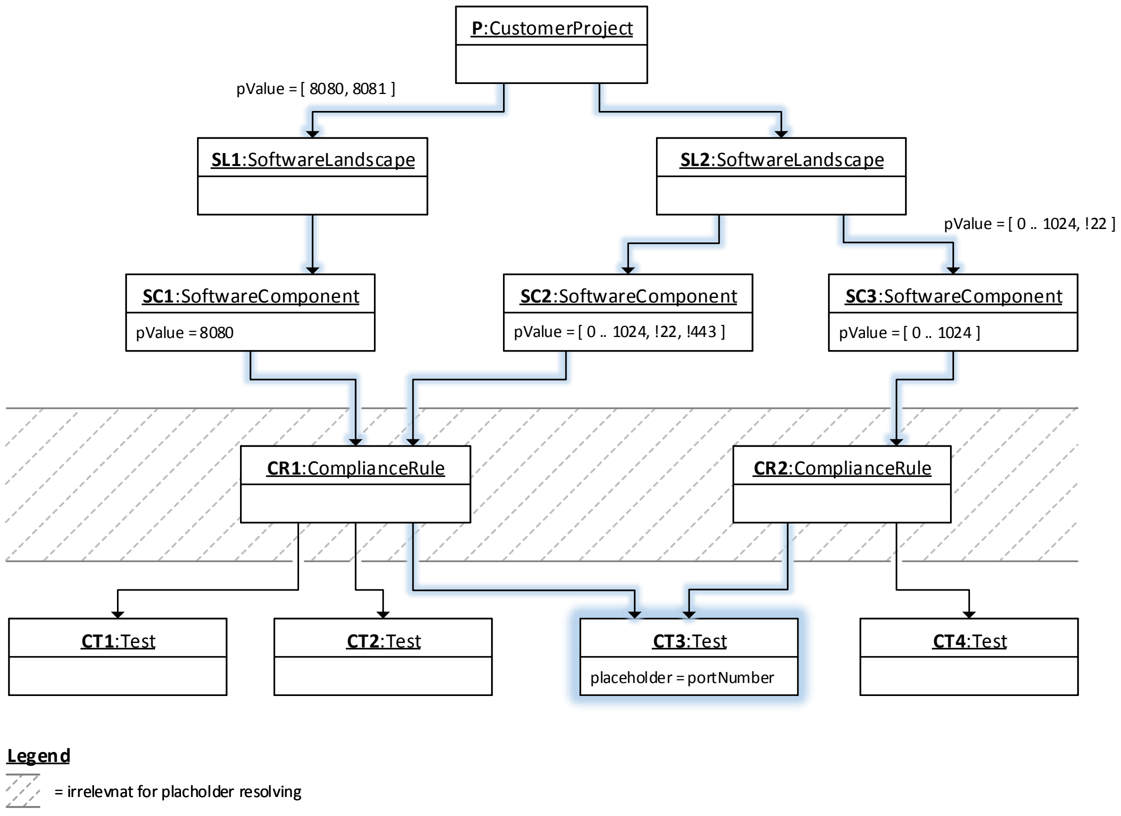


Figure 6.8.: An exemplary partial model defining a sample *CustomerProject* P crafted on basis of the defined *CMT meta model*. It contains the central its aspects, i.e., *Artifact*, *Compliance* and *Test*, and illustrates the (finally) derivable tests. Highlighted paths correspond to all possible derivation paths for *ComplianceTest* CT3. Each path has, if needed, an annotated *CMT-Exp* to define or relax the *PlaceholderValue* on this unique path.

Step 1 - Derivation Starting at the *CustomerProject* P, all possible *ComplianceTests* need to be derived, i.e., simply traversing down the data-tree. Since one test, e.g. CT3, can be reached through multiple paths, the most accurate path needs to be stored for a unique (context) identification. For example, the rightmost derivation yields the path-id `p.sl2.sc3.cr2.ct3`. After visiting a leaf-node, i.e., *ComplianceTest*, the backwards traversal (revisiting all parents) is used to determine possible *PlaceholderValues*, i.e., *Determination*.

Step 2 - Determination This step accomplishes the *PlaceholderValue* determination, which is conducted during the backwards traversal. When (re-)visiting a parent artifact entity all defined *PlaceholderValues* are checked. If one parent entity defines a value for the *Placeholder*, it overwrites all already existing *PlaceholderValues* on this path. This step can result in a more general definition (*generalization*) or a more specialized definition – depending on the *CMT-Exp*. However, all previous defined *PlaceholderValues* are overwritten and, explicitly, not extended to preserve the principle of locality defined by Denning et al. [Den05]. Otherwise, if inheriting all previous defined *PlaceholderValues*, changes to lower-level entities would unintentionally also change the parent's behavior/context. This effect would dissent the *Generalization Hierarchy* introduced in the conceptual *meta model* (cf. Section 6.3).

An exemplary overwrite is illustrated by *SoftwareLandscape* SL2 and its right outer derivation path, which overwrites the *PlaceholderValues* defined on *SoftwareComponent* SC3. Since the context path for SL2 (`p.sl2`) is more general than the one for SC3 (`p.sl2.sc3`), this proceeding is named *generalization*.

Step 3 - Instantiation In the last step, the concrete test instantiation is performed. For each resolved *PlaceholderValue* (through *Determination*) one *test case* is generated. Afterward, the generation process has to evaluate the defined *CMT-Exp* and for each value of the expression on a single test should be created and attached to the test case. The chosen example (*ComplianceTest* CT3) yields three test cases. Furthermore, for the test case of context B (cf. Table 6.1) 1023 tests are generated and attached due to the defined expression.

With the presented concept and the ability of *Test Genericity* we extend the approach of *Data-Drive-Testing* [Mes07] towards an "*Generative-Data-Driven-Testing*" attempt. More precisely we substitute the *static* test data source with an *dynamic* derivation process as defined above. To this end, we state that adopting this concept to our defined domain, i.e., the *Product*, on is able to model different software products, their components and compliance requirements in a modular and reusable way.

Blocking

The fundamental and straightforward idea behind *Blocking* is to support finalization of *PlaceholderValue* at certain stages (*Test*, *Compliance* or *Artifact*) in the model, i.e., to prevent further definition or overriding in a higher context. Thus, this concept yields the contrary functionality as proposed by the *Whitelisting* concept.

However, this feature yields, even more, test reusability since one can write as general tests as possible and specify their semantic meaning on *Compliance* level. For example, writing a single *Test* while ensuring different file modes is possible, i.e., defining a *Placeholder* for the specific file mode property. By applying the concept of *Blocking*, one is now able to reuse this test and repeatedly instantiate this *Placeholder*, e.g., with "read" to check if a certain file is readable. This approach results in a more accurate and atomic compliance requirement which only checks whether a file is readable. The adaptation of such a compliance requirement requires less knowledge about the concrete test functionality and thus improves the applicability of *Test Genericity.*

To strength the *continuousness*, e.g., using the *CMT* in an automated manner, we introduce further tooling as part of the *Execution* activity next.

6.4.2. Inspection – Test Execution and Simulation

] To satisfy the second phase of the *Continuous Compliance* process, i.e., *Develop*, and its defined *Inspection* activity the *CMT* needs to be capable of *generating* and *executing* the modeled compliance requirements. As we want to encourage a high adaptability of our tooling, we propose the creation of external *Test Plugins* which rely on existing test frameworks, e.g., *InSpec* or *OpenSCAP*.

The central task is to derive and instantiate all modeled compliance tests for an *SUT*. These tests then need to be transformed and executed. Next, the *CMT* has to collect all outcomes to support further evaluation and reporting. However, as the fundamental functionality needs to be performed by the appropriate implemented *Test Plugin*, we do not define any concrete concept and rather rely on existing functionality of the used test framework.

Simulation Besides the actual execution of *Compliance* on an *Environment*, we propose the idea of *Compliance Simulation.* Given the previously defined modeling approach, it is possible to *simulate* the compliance testing using the model itself. For example, the model could be instantiated and validated against certain constraints proving its correctness. More precisely, we purpose the concept of *Compliance Validation* and *Placeholder Validation.*

Compliance Validation should validate if each modeled *ComplianceRule* is at least *implemented* by one *Test.* Otherwise a compliance test run would yield improper results as not every demand *Compliance* could be ensured.

Placeholder Validation should check if each *Test* is instantiable, i.e., each used *Placeholder* has at least one *PlaceholderValue* definition. Otherwise, when executing unresolved *Tests*, it would lead to unexpected behavior in the *Test Runner*.

External Accessibility To further support the integration into an external automation system, we introduce the *CMT - Command Line Interface (CMT-CLI)*.

The *CMT-CLI* represents a command line interface which is able to remotely control the *CMT*. To this end, the tool or script should be able to search and trigger (for execution) all available compliance tests. As a consequence, the *CMT-CLI* needs to be able to authenticate against the *CMT* and should be capable of communicating with the *CMT* in a certain way. Furthermore, it requires the ability to fetch all available results and outcomes once a compliance test run has terminated. Summarized, the *CMT-CLI* should cover all fundamental functionality, i.e., searching, listing and execution of compliance tests, and simplify the remote interaction.

Utilizing the *CMT-CLI* in conjunction with an automation system, e.g., Jenkins, performing compliance testing in an *continuous* manner is desirable, i.e., *Continuous Compliance Testing*.

6.4.3. Reporting

To accomplish the last outlined requirement for a tooling support, i.e., *Reporting*, we present metrics based on the previously gathered test results. As metrics are an important factor in making sound decisions on the efficiency of security (or similar) operations [Jan10], we claim that such reports and their metrics support the management and reduce the problem of *Management Inexperience*. To this end, we present and elaborate on different *Key Performance Indicators (KPIs)* used for visualizations and outline the concrete concept towards a calculation of the current violation score, i.e., the *Maximum Compliance Violation*.

To report the tests results in a simplified manner, but as fine grained as possible, we suggest three different KPIs, i.e., the *Compliance Index*, a *Compliance Violation* and *Derivation Graph*.

Compliance Index represents the relation between all passed and failed compliance. This relation should be visualized by a simple chart which allows a quick and simple interpretation of how compliant the *ApplicationSystem* is.

Compliance Violation represents the violation classification of failed tests. It should interactively visualize the three different violation states (Low, Moderate and High) and indicate the current violation score based on the *Maximum Compliance Violation* of all failed tests.

Derivation Graph shows a partial model, i.e., *Product*, *Compliance*, and *Test*, of the meta-model domain including all derivation paths starting with the top-level domain, i.e., the *Product*.

This visualization helps to comprehend all modeled relation and inherited tests used for this particular test execution. Furthermore this visualization allows to track the inter-model relation easier and indirectly leads to a fundamental documentation of the modeled *Product*.

The detailed *Maximum Compliance Violation* calculation required for the *Compliance Violation* reporting is described with the help of a fictional scenario in the following.

Maximum Compliance Violation To provide a *Compliance Violation* report, a *somehow* calculated violation value for each test is required. Due to the general test derivation, which is the of *Test Genericity* and *Whitelisting* (cf. Section 6.4.1), a context-sensitive calculation is necessary. The resulting *violation score* for a single test should represent all its inherited impact factors of its superior modeling elements, i.e., those factors defined by *Compliance* and *Artifact* parent entities (cf. Section 6.3). Furthermore, the *Maximum Compliance Violation*, should only include the violation score of those tests which failed. To this end, an score of 0 (zero) represents a compliance test with only valid tests.

In Table 6.2 all factors, which are part of the particular test violation score calculation, are explained. The foundation for the derived values are those properties defined in the domain model. The utilized values in Table 6.2 are derived from a fictional scenario *CustomerProject P* and its modeled dependencies shown in Figure 6.9. The final outcome (*90%*) represents the *Maximum Compliance Violation* for *P*.

To specify the calculation, we define a set of equations next, which are sufficient to implement the defined *Maximum Compliance Violation*.

Equations 5.2, 5.3 and 5.4 correspond to the maximum value of all possible *impacts*, *significances* and *types* along the derivation path. Furthermore, the type categorization (*REQUIRED, RECOMMENDED, OPTIONAL*) is mapped on thirds ($\frac{1}{3}$) using Equation 5.1. The *Compliance Violation* (CV) per test is expressed by Equation 5.6. It results from adding the weighted share to its basic offset (*Type*). The final *Maximum Compliance Violation* calculation, expressed in Equation 5.7, is derived by taking the maximum over all *CV*. Hence, only the highest violation score is reported.This is justifiable as we only intend to present the *Compliance Violation* foreseeable due to missing compliance.

Result Interpretation To facilitate a result interpretation for our presented *Maximum Compliance Violation* metric a mapping or similar is required. To this end, we provide an interpretation function from the numeric scale to an ordinal scale, i.e., a measurement scale on which the subjects can be compared in order [Kan02]. In detail, we map the numerical scale $0 - 100$ to the ordinal scale $Low < Medium < High$ defined as follow:

$$Interp(cv) = \begin{cases} Low, & \text{if } 0 \leq cv < 33 \\ Medium, & \text{if } 33 \leq cv < 66 \\ High, & \text{if } 66 \leq cv \leq 100 \end{cases}$$

	$Test_A$		$Test_B$		$Test_C$	
Artifcat	Significance		Significance		Significance	
Project	7		7		7	
Landscape	6		5		3	
Component	8		3		–	
Significance(T)	8		7		7	
Compliance	Impact	Type	Impact	Type	Impact	Type
Profile	2	Req	–	–	10	Req
Rule-Set	4	Opt	4	Opt	–	–
Rule	5	Rec	5	Rec	6	Rec
Impact(T)	5		5		10	
Norm(Type)		$66\frac{2}{3}$		$33\frac{1}{3}$		$66\frac{2}{3}$
Compliance Violation (CV)	$46\frac{2}{3}$		45		90	
Maximum CV	90					

Table 6.2.: Exemplary violation score calculation per tests and the overall *Maximum Compliance Violation* result. Values used for calculation are based on a fictional scenario shown in Figure 6.9. For the sake of simplicity we assume that all three tests cases failed. *Req, Opt* and *Rec* are abbreviations for the *ClassificationTypes* introduced in the CMT domain model (cf. Figure 6.7)

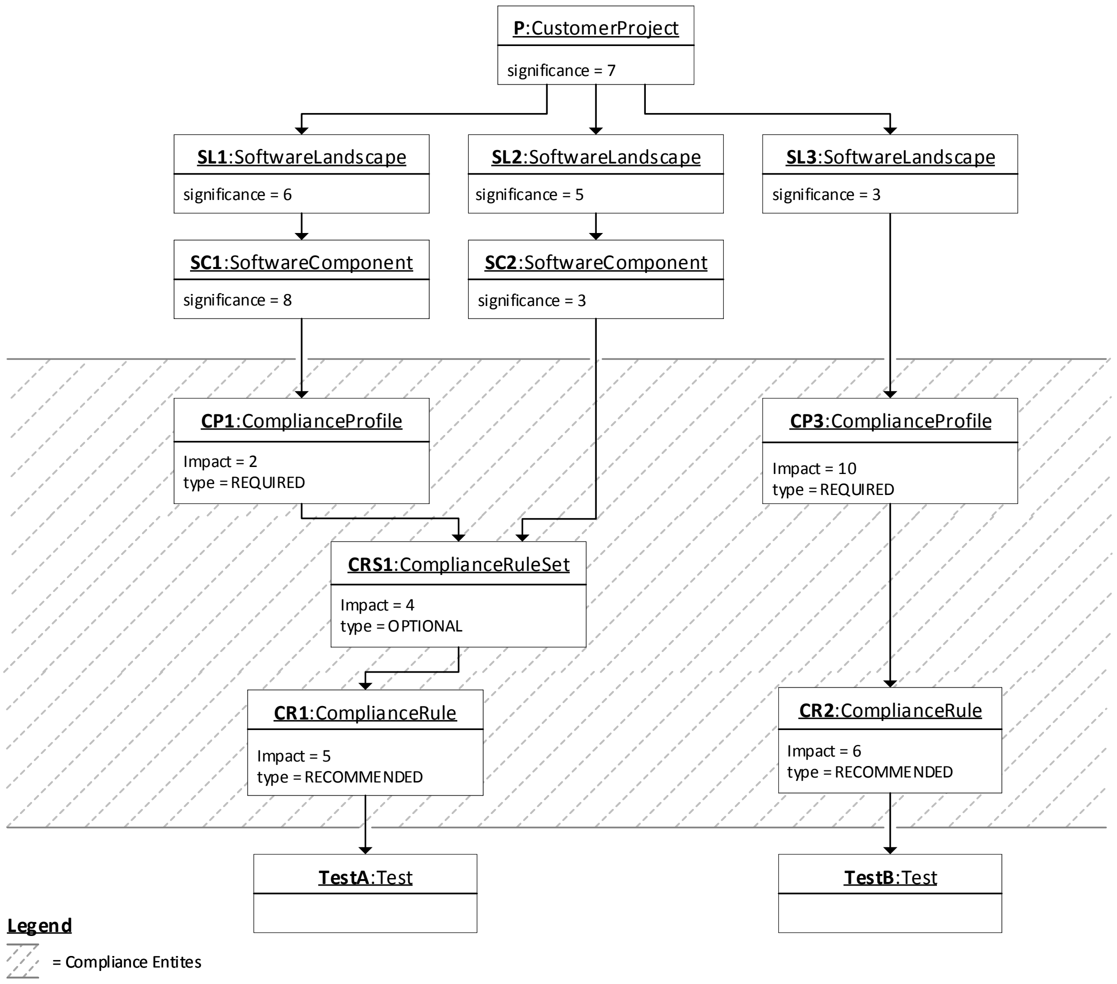


Figure 6.9.: An exemplary partial, incomplete model defining a sample *Customer-Project* P with all its derived *Artifacts*, *Compliances* and *Tests*. Each entity shows their associated impact on the violation score, i.e. *significance*, *impact* and *type*. As a result of the derivation process, three different *ComplianceTest* instances will be generated for the *CustomerProject* P, i.e., `P.SL1.SC1.CP1.CRS1.CR1.TestA`, `P.SL2.SC2.CRS1.CR1.TestA`, and `P.SL3.SC3.CP3.CRS3.CR3.TestB`

.

$$C_T = \text{Set of all parent-related Compliance entities of Test T}$$

$$A_T = \text{Set of all parent-related Artifact entities of Test T}$$

$$T_{CP} = \text{Set of all Tests for one Customer-Project CP}$$

$$NormType(Type) = \begin{cases} 0, & \text{if } Type = Optional \\ 33\frac{1}{3}, & \text{if } Type = Recommend \\ 66\frac{2}{3}, & \text{if } Type = Required \end{cases} \tag{6.1}$$

$$Impact(T) = max_{c \in C_T}(Impact(c)) \tag{6.2}$$

$$Significance(T) = max_{a \in A_T}(Significance(a)) \tag{6.3}$$

$$Type(T) = max_{c \in C_T}(NormType(c)) \tag{6.4}$$

$$TypeShare = \frac{1}{3} \tag{6.5}$$

$$CV(T) = Type(T) + TypeShare * Impact(T) * Significance(T) \tag{6.6}$$

$$MaxCV(CP) = max_{t \in T_{CP}}(CV(t)) \tag{6.7}$$

However, we intentionally do not claim this calculation as a *Security Metric* for the *ApplicationSystem* due to following circumstances. First and foremost, research so far does not yield a sound definition regarding *Security Metrics* although it constitutes one major research challenge in information security [Jel00; Sav07]. Further, developing generally applicable security metrics is almost impossible as the organization's objectives regarding security can widely vary, which impacts the interpretation and validity of metrics [Jel00]. To this end, we remain with "'simple" KPIs, i.e., *Compliance Index*, *Test Amount*, and *Maximum Compliance Violation*,to support the management in making further decisions.

6.5. Summary

Finally, in Chapter 6 we explained the practical adaptation of the general process model and introduced the concept of *Continuous Compliance Testing*. We described various requirements for a tool supporting the process and presented our novel ideas for the *Compliance Management Tooling* in Section 6.2. To conclude this definition, we defined a meta model which contains all necessary aspects to successfully create a tooling for our process model, i.e., the *Software Meta-model* (cf. Section 6.3).

We successfully defined a more technical motivated process model, i.e., *Continuous Compliance Testing*, and requirements for a *Compliance Management Tooling* to support the adaptation. Afterward, we present further, more technical and engineering related concepts, which are relevant to realize such a tooling and respect the significance of entity (*Test*, *Compliance* and *Artifact*) *Reusability* (cf. Section 6.4.1). Furthermore, we described technical concepts to allow test *Execution* (cf. Section 6.4.2) and *Reporting* (cf. Section 6.4.3) about the gathered test results.

7. Software Architecture & Realization

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

LARRY WALL

Contents

In this chapter,we describe a proof-of-concept realization of our previously defined tool support to enable *Continuous Compliance Testing*.

First and foremost we elaborately explain our taken approaches, ideas and applied patterns to create a (re-)usable *Compliance Management Tool* in Section 7.2. We start with a precise characterization of the fundamental technology stack, followed by more detailed description of the implemented *Compliance Model* in Section 7.2.2, as a realization of the *Domain Model* (cf. Section 6.3). Afterward, we present our implementation of the two central functional concepts, i.e., *Test Genericity* and *Whitelisting*. Finally, in Section 7.2.5 we present the essential patterns and techniques applied to implement compliance test generation and execution with the help of an exemplary *Test Plug-in* (cf. Section 7.1) implementation, i.e., `InSpecTestRunner`.

7.1. Software Architecture

Considering the previous mentioned meta model and the further technical motivated concepts, we present a resulting software architecture for our novel framework. It follows a classical three-layer architecture style. Similar to the three main-layered architecture blueprint submitted by Fowler et al. [Fow02], we likewise relay on three semantical different layers as illustrated in Figure 7.1. First, on the bottom, a *Data Source* layer is used to cover all transactional communication with data-related system, e.g., database management system. It consists of the *Repository* component and the *RDBMS*, *VCS*, and *File*- system.

Next, in the mid layer, we applied the pattern of a *Domain* or *Business* layer, i.e., grouping all business related functionality and components. This layer is composed of the *Domain*, *Core*, *Adaptation*, *Test*, *Test Plugin*, and *Reporting* components.

The top-most layer represents our *Presentation* layer and provides all services which are required for information derivation and manipulation. Furthermore, it includes the components used to offer user interaction with the system. This layer consists of the *Communication* and *User Interface* component.

In the following, we present all previous components, their central functionality, their provided service and interaction with other components in detail. For that, we peruse the architecture blueprint illustrated in Figure 7.1 from bottom to top and simultaneously from left to right.

Repository Component provides all necessary functionality required to store and fetch the data from a rational database management system (RDBMS). To this end, this component has to adopt certain interfaces from the chosen RDBMS.

Domain Component enfolds the four central aspects required for modeling the *SUT* and their related data-objects. Its central functionality is to provide all domain related data as well as its management. For that reason, it is strictly aligned with our meta model presented in Section 6.3 and contains the main aspects *Artifact*, *Compliance*, *Test*, and *Environment*. Additionally, this component provides interfaces to the *Communication* and *Core* component to allow those to use and operate on the data.

Core Component is the central component and provides all necessary functionality to apply the wanted business-logic, i.e. testing or reporting, on the stored domain entities. To this end, it allocates further components, such as *Filter* and *Traversal*. As their names indicate they supply basic functionality to *filter* entities in a model and to *traverse* the whole data structure moreover, e.g., to derive all linked *Tests* for a *Customer Project*. Furthermore, the *Core Component* includes components which model the *Job* and *Placeholder* domain. We deliberately decided to separate those "domain" entities from the *Domain Component*. As they only rely on the *Domain Component* and do not contribute the functionality of the overall components we split those concerns into distinct features (cf. "Separation of Concerns" [Dij82])

The *Core Component* builds upon the provided functionality of the *Domain* and

Repository component. Moreover, it provides interfaces to its functionalities for the *Communication*, *Adaptation* and *Test* Component.

Adaptation Component represents the “anchor” component to trigger the compliance test run. It contains two separated components *Execution* and *Simulation.* One component wraps the functionality to initiate (execute) a full compliance test run and the other provides techniques for simulating a compliance test. To this end, the *Execution* component additionally depends on interfaces offered by the *Test Component.* Both rely on the *Core Component* to access central functionality, such as *filter* and *traverse.* Summarized this component delegates certain duties during compliance test execution/simulation, e.g., creating *Job* entities.

The only interface provided by this component is associated to the *Communication Component* to allow compliance test execution/simulation via an external interface.

Test Component focuses all duties required for compliance test execution and furthermore their generation. It enfolds four main components, i.e., *Runner*, *Evaluation*, *Transformation*, and *Export.* The key component is *Runner* which should be used to execute the compliance test run. For this propose the executable test artifacts needs to be generated with the help of the *Transformation* component and evaluated with the aid of functionality provided by the *Evaluation* component, afterward.

Although this component includes various components to provide its functionality it only acts as a *template* and just defines the particular function flow. To this end, it should remain as a *semi-finished* component [Wol94]. The primary functionality, e.g., test execution, needs to be provided by one or more *Test Plugins* and the implementation of the appropriate interface. This separation generates a *Hot Spot* (cf. [Wol94]) and turns our overall software architecture into a *Blackbox-Framework.* As outlined by Fayad et al. a Blackbox frameworks support extensibility by defining interfaces for components that can be plugged into the framework via object composition [FS97]. In our architecture, this feature is realizable through *Test Plugin Components* which are described next.

Test Plugin Component warps the interfaces required for *external* plug-in development. The *Test Plugin* itself implements the actual functionality to execute, evaluate, transform and export a test-case. It only offers its interfaces to the *Test Component* and is completely independent of the overall architecture. As argued above this plug-in mechanism realizes a blackbox framework accordingly to Fayad et al. [FS97].

Reporting Component provides functionality for generating a report based on a finished compliance test run. This component encapsulate to core functions *Reporting* and *Compliance Violation Calculation* which we outline detailed in *Reporting* concept (cf. Section 6.4.3). To receive the required information it relies on the interfaces of the *Evaluation* component provided by *Test Component.* Since this component does not offer certain functionality requested by the framework itself it only provides one interface which is associated with the *Communication* component to allow external retrieval of the reporting results.

Communication Component comprises the framework core, i.e.,*Domain*, *Core*, *Adaptation*, *Test*, and *Reporting* from the presentation layer, e.g., the *User Interface* component. Thus, it provides three central resources which are all accessible through any communication method, e.g., REST or similar. Those components act as abstraction and delegation towards the main business-logic. To this end, the *Communication Component* does not introduce new functionality.

User Interface Component interacts with the *Communication Component* only and enfolds different presentation technique for user interaction. We provide two main components, i.e., *Web Application* and *Command Line Interface*, which should cover two essential interaction possibilities. First and foremost the operation through a full-featured webinterface and furthermore a simplified interaction through systems command line, e.g., to allow integration or interaction with headless systems[1].

Summarizing it is to say that the finally proposed *Software Architecture* blueprint, illustrated in Figure 7.1, enfolds the fundamental meta models, concepts, and functionality required to realize the proposed *Compliance Management Tooling* as supporting tool for *Continuous Compliance Testing*. Furthermore, we have outlined the central components which turn the architecture and its resulting system into a conceptual *Blackbox Framework* with certain *Hot Spots* for further extension (cf. definition of *Test Component*).

Based on this architecture blueprint and its implicitly defined data model we present a possible realization of this tooling next. We outline the most relevant aspects needed to accomplish the previously outlined *engineering* concepts.

[1]A headless system is a computer that operates without a monitor, graphical user interface (GUI) or peripheral devices, such as keyboard and mouse. `http://internetofthingsagenda.techtarget.com/definition/headless-system`

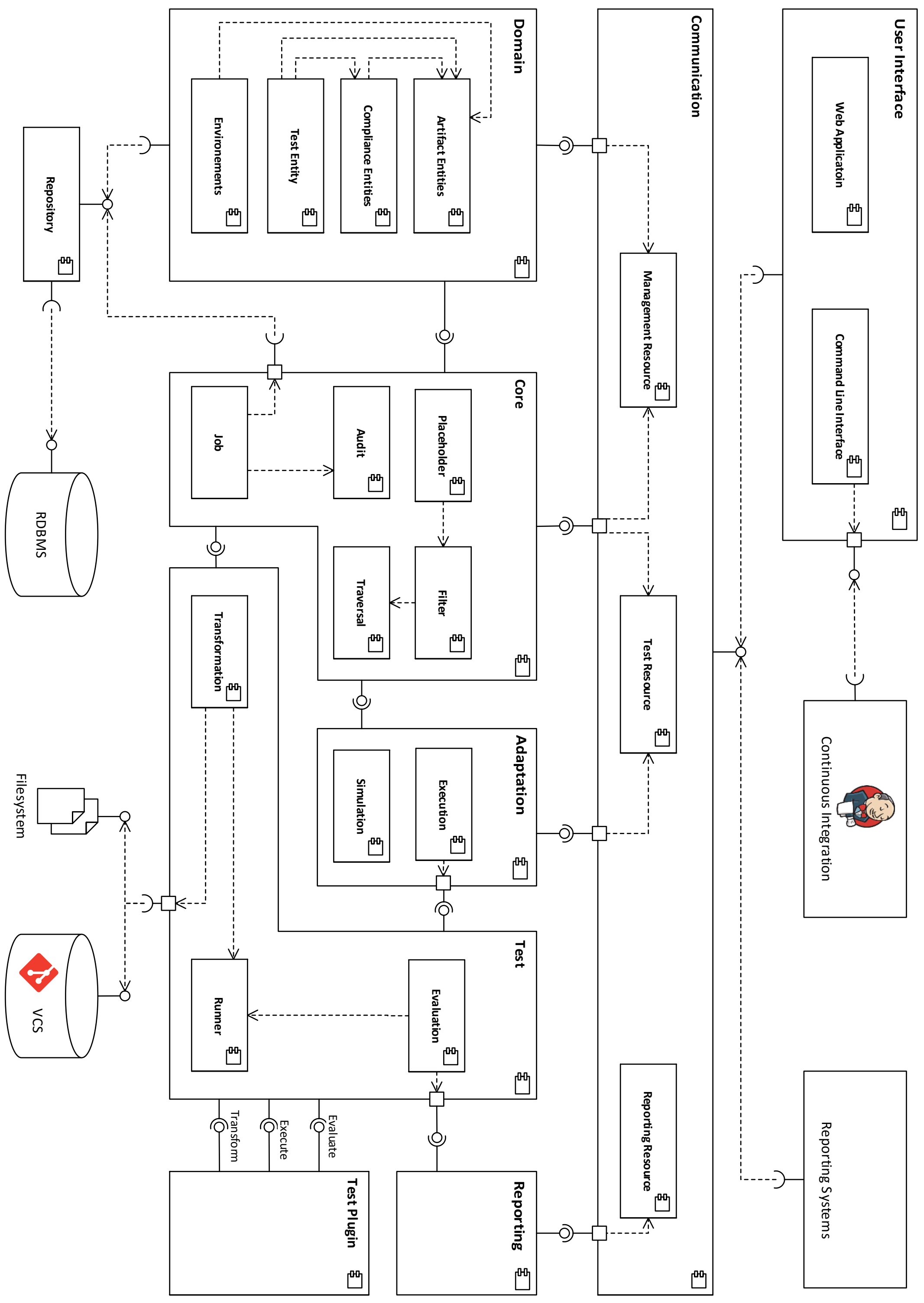


Figure 7.1.: *Compliance Management Tooling* Software Architecture Blueprint – A three-layered architecture is illustrated and all relevant components as well as their interfaces are shown. Furthermore, adaption to external systems like Jenkins is sketched.

7.2. Configuration Management Tool

Like specified in the software architecture blueprint (cf. Section 7.1) the CMT should follow a three-tier architecture. We omit a detailed description of the lowest layer, i.e., the database layer since those are nowadays easily realizable with production-ready components. To this end, we only focus the *Domain* layer (afterward also referred to as *Backend*) and the *Presentation* (afterward also referred to as *Frontend*) in greater detail.

7.2.1. Technologies

The fundamental technology stack for the CMT is (indirectly) specified through the applied generator JHipster, which allows generating an extensive and executable bedrock for web-based applications including a Java backend and Angular frontend quickly [DS16].

We intentionally decided to use a generator for the fundamental code and functionality to decrease the time effort spent on project setup, configuration and implementation of *common* functionality like database access and so forth. The resulting technology stack is outlined next, distinct by its field of application.

Backend The backend of our tooling is completely based on Java. Given JHipster, the main components and frameworks used in our realization are part of the *Spring* framework eco-system. For example, *Spring Boot* is used to simplifying the application configuration and *Spring Security* is included to provide customizable authentication and access-control [DS16; Piv02]. Furthermore, the *Spring Data JPA*, i.e., enhanced support for the *Java Persistence API*, for data management and the *Spring MVC REST* in conjunction with *Jackson* are applied. To this end, the *CMT* uses *Representational State Transfer (REST)* to implement the *Communication* component.

Utilizing these frameworks and libraries further assure continuous use and development of our first, proof-of-concept, implementation, since these libraries are widely known and still under active development.

Summarized on can say, that the overall backend, i.e., the data management, core business application logic and the communication endpoint, are fully based on *Java Platform, Enterprise Edition (J2EE)* in conjunction with the *Spring Framework*.

Frontend Similar to the backend the main frontend technologies are addicted to those given through the JHipster generator. *Angular* in version 4 founded by Google [Goo10] is used to realize the *Web Interface* component as part of the presentation layer. Its fundamental technology is *TypeScript* by Microsoft a superset of JavaScript, e.g.,introducing type-safety, that compiles to plain JavaScript. Using *Angular* allows to follow a modular design approach in the frontend as well. With *TypeScript*, i.e., the ability to define classes, interfaces, and inheritance, we were able to replicate the data structure defined in the backend. This makes the comprehension of the frontend component even easier.

To realize a *state-of-the-art* visualization and interaction, we rely on *HTML5* and furthers frameworks like *jQuery* and *Bootstrap*. Additionally, we utilized the *D3.js* library to implement a visualization of the full model and its siblings as part of the reporting interface (cf. Section 7.4).

To summarize, the applied frontend techniques engender a *Single-Page Application (SPA)* which conforms to one trend in the field of web development.

As barely described above the *CMT* technology stack is comprehensive but allows a very straightforward and modern development process. Because we only intend to prove our outlined concept with the help of a *proof-of-concept* implementation, we skip further evaluation or classification of the used technologies at will.

Our central, most important, attribute was a simple application bootstrapping at high velocity to spend as much time as possible on business-logic implementation. This was reached with the adaptation of JHipster.

7.2.2. Compliance Model

To implement the data structure aligned with domain model presented previously, we used the *JHipster Domain Language (JDL)*. With it, we were able to model all required entities and their properties in an ordinary way. Listing 7.1 outlines the self-explaining definition of the *ComplianceRule* entity. However, JDL is not yet capable of model-inclusion or definition abstract entities, whereas we needed to include the general properties defined through the abstract *Compliance* entity (cf. Figure 6.7) as well.

```
entity ComplianceRule {

    // Abstract Metadata
    title String required,
    description String,

    // Compliance Metadata
    type Classification required,
    impact Integer required min(0) max(10)
}
```

Source Code 7.1: Model defintion for domain entity ComplianceRule

Furthermore, using JDL, we were able to describe various kinds of relation likewise. The relationship model in Listing 7.2 defines *Many-to-Many* relationships between `ComplianceRule` and other entities. Despite the defined relations in our concept, we had to fall-back to *Many-to-Many* relations since JHipster does not yet provide the ability to model and generate unidirectional *One-to-Many* relations [DS16]. For the sake of simplicity and possible later revision of our model, we adapted the most-general *Many-to-Many* relationship for all entity relations. If required one could limit those with hand-coded context conditions on model level.

```
relationship ManyToMany {
    ComplianceProfile{rule} to ComplianceRule{profile},
    ComplianceRuleSet{rule} to ComplianceRule{ruleSet},

    ComplianceRule{test} to ComplianceTest{rule}
}
```

Source Code 7.2: Relation defintion for domain entity ComplianceRule

With these data models were able to generate a full *Create, Read, Update and Delete (CRUD)* application including a backend and frontend easily. For each entity, the JHipster generates a *database table*, a *liquibase change set* to perform database migrations, a *JPA entity*, a *Spring Data JPA Repository* and a *Spring MVC REST Controller* which implements the basic CRUD operations. The resulting classes and interfaces – six in total – of the exemplary *ComplianceRule* entity (cf. Listing 7.1, Listing 7.2) is depicted in Figure 7.2. The two major classes, which are relevant for later adaptation, are `ComplianceRule` and `ComplianceRuleService`. The former one implements the plain (data-) model, e.g., the accurate mapping of the defined entity in Listing 7.1. The CRUD functionality is implemented through the latter one.

To conclude, using JHipster, we do not had to implement all *database* related operations and can construct our own, business-logic on basis of this.

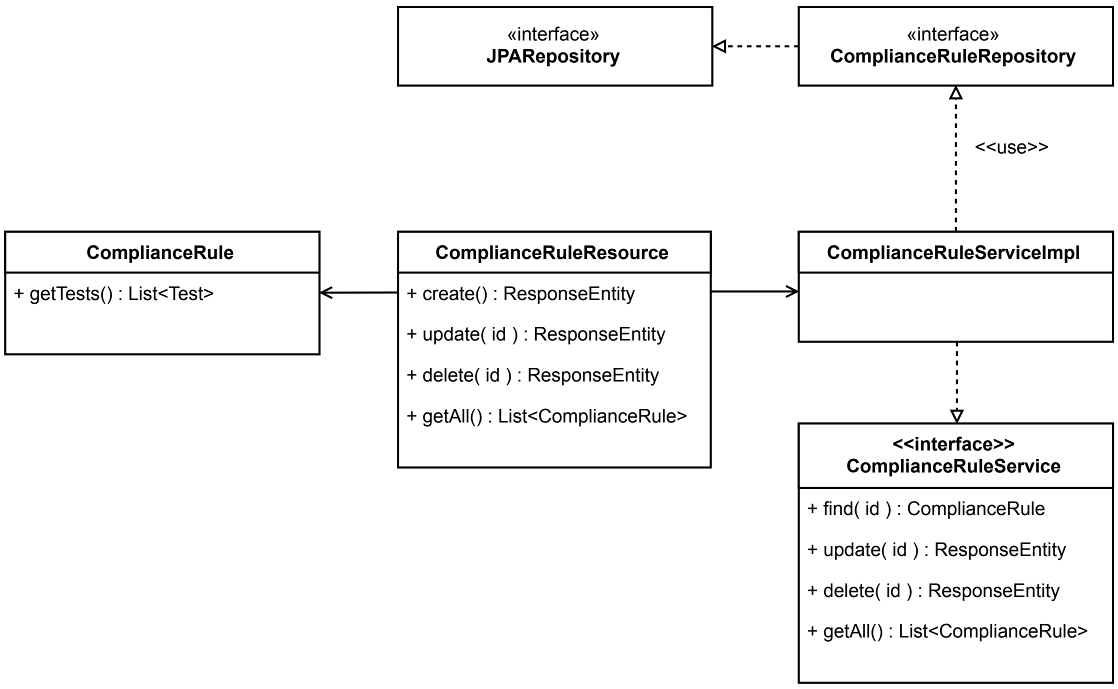

«interface»
JPARepository
«interface»
ComplianceRuleRepository
<<use>>
ComplianceRule
+ getTests() : List<Test>
ComplianceRuleResource
+ create() : ResponseEntity
+ update(id) : ResponseEntity
+ delete(id) : ResponseEntity
+ getAll() : List<ComplianceRule>
ComplianceRuleServiceImpl
<<interface>>
ComplianceRuleService
+ find(id) : ComplianceRule
+ update(id) : ResponseEntity
+ delete(id) : ResponseEntity
+ getAll() : List<ComplianceRule>

Figure 7.2.: UML class diagram illustrating the central *interfaces* and *classes* generated by the *JHipster* for the *ComplianceRule* entity.

However, due to the missing adaptation of Genericity, i.e., through non-existing abstract classes or interfaces, the resulting data structure is not modular nor supports it an appropriate adaptation. For example, to search or filter certain entities in a model, e.g., Placeholder extraction, a significant amount of type-specific code is required since we could not rely on super-classes or interfaces. To this end, we decide to introduce additional, hand-coded, abstraction for each entity (cf. Figure 7.3). The highest level of abstraction is given through the implementation of `Entity` interface, which provides some basic functionality. Slightly more specific are the interfaces `ComplianceEntity`, `PlaceholderValueDefiningEntity` and `ArtifactEntity`, each extending the superior interface *Entity*. Depending on the entity functionality it inherits one or two of these interfaces. For example, the `SoftwareComponent` inherits the `ArtifactEntity` interface as its more abstract type is an *Artifact*. Furthermore it inherits the interface `PlaceholderValueDefiningEntity`, because it is used to define a value for derived placeholders. The design and implementation of this abstraction, their general functionality and inheritance relations are shown in Figure 7.3.

To make use of the added value, i.e., abstraction, and improve easy data handling (traversing and filtering) we further implemented the *Visitor-Pattern*. This pattern allows us an later definition of further operations without changing the classes of the elements on which it operates [Gam+95]. Due to *double-dispatch*, i.e., double dispatching the call until the actual operations gets executed which is indicated by `accept(EntityVisitor v)` method, every operation is executed in the actual object context.

The adaptation of the *Visitor-Pattern* and its concrete realizations its illustrated in Figure 7.4. To specify the responsibility for traversing the object structure, which is not clearly stated by the *Visitor-Pattern*, i.e., it could be realized by through the visitor, the object-structure itself or an additional iterator [Gam+95], we implemented the fundamental traversal functionality with an abstract `BasicEntityVisitor`. Thus we put the responsibility for traversing within the visitor as well.

The `BasicEntityVisitor` templates the `handle` methods for each entity and thus implements the `EntityVisitor` interface. A default handle methods performs the visit (`this.visit(entity)`), specific traversal (`this.travers(entity)`) and terminates the visit (`this.endVisit(entity)`). These nested methods are scaffold by the `BasicEntityVisitor` as well and reduce the effort required to implement new operations. On the basis of this, we created three specific visitors which realize a variety of function.

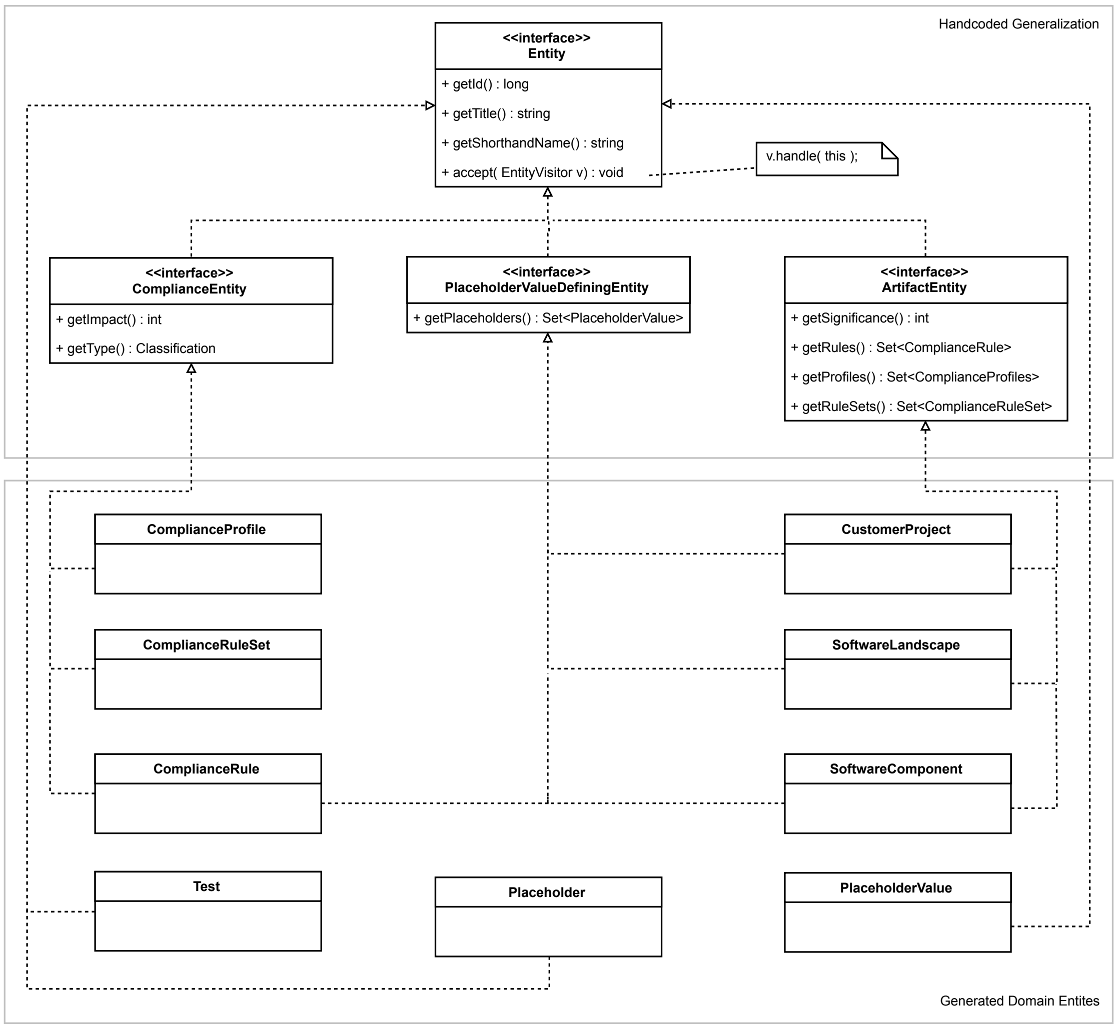
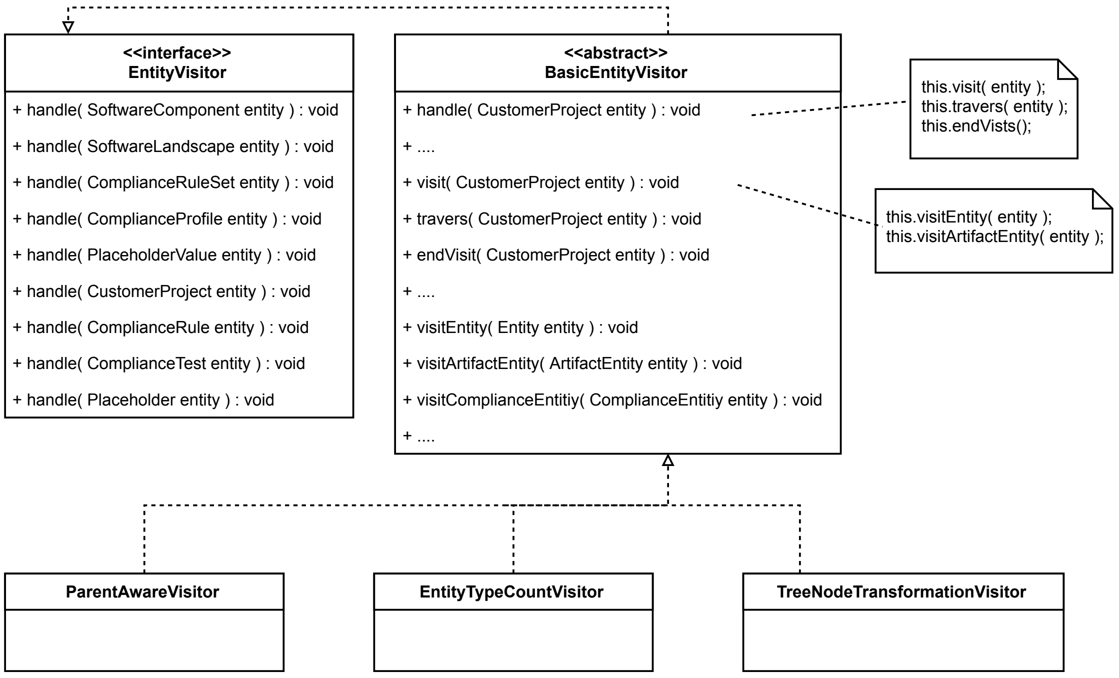


Figure 7.3.: UML class diagram illustrating all generated entities (bottom) and all hand-coded interfaces (top). It emphasizes the achieved simplicity, i.e., describing all entities with three distinct interfaces. The most general interace, i.e., *Entity*, indicates the entry-point for various *EntityVisitor*, i.e., `accept(EntityVisitor v)`



<<interface>>
EntityVisitor
+ handle(SoftwareComponent entity) : void
+ handle(SoftwareLandscape entity) : void
+ handle(ComplianceRuleSet entity) : void
+ handle(ComplianceProfile entity) : void
+ handle(PlaceholderValue entity) : void
+ handle(CustomerProject entity) : void
+ handle(ComplianceRule entity) : void
+ handle(ComplianceTest entity) : void
+ handle(Placeholder entity) : void
<<abstract>>
BasicEntityVisitor
+ handle(CustomerProject entity) : void
+ ....
+ visit(CustomerProject entity) : void
+ travers(CustomerProject entity) : void
+ endVisit(CustomerProject entity) : void
+ ....
+ visitEntity(Entity entity) : void
+ visitArtifactEntity(ArtifactEntity entity) : void
+ visitComplianceEntitiy(ComplianceEntitiy entity) : void
+ ....
this.visit(entity);
this.travers(entity);
this.endVists();
this.visitEntity(entity);
this.visitArtifactEntity(entity);
ParentAwareVisitor
EntityTypeCountVisitor
TreeNodeTransformationVisitor

Figure 7.4.: UML class diagram illustrating the fundamental *interfaces* and *classes* implemented to adopt the *Visitor* pattern. A abstract implementation is given by *BasicEntityVisitor*.

ParentAwareVisitor serves as visitor and traversal implementation with additional awareness of all parent elements. To do so it "pushes" the current entity to a stack during the `visit` methods and "pops" the topmost entry during `endVisit`.

This very straightforward implementation, it requires only four line of relevant code as complied in Listing 7.3, yields an enormous key functionality for further realization, e.g., placeholder derivation or test code generation. The most notable benefit given through this visitor is the derivable context for each entity, i.e., iterating the parents dequeue yields the unique entity context.

EntityTypeCountVisitor implements a simple *counting* functionality to receive the amount of used or inherited entities. This visitor is useful for metric calculation on a model. For example, this visitor is used to count the amount entity utilization as part of the reporting (cf. Section 7.4).

TreeNodeTransformationVisitor implements a transformation function to transform the object structure from a *Polyhierarchy* to a *Monohierarchy* structure, i.e., tree-like structure. We use this visitor to generated the data structure required for further visualization or documentation.

```
public class ParentAwareVisitor extends BasicEntityVisitor{

    protected Deque<Entity> parentEntities;

    public ParentAwareVisitor() {
        this.parentEntities = new ArrayDeque<>();
    }

    @Override
    public void visitEntity(Entity entity) {
        super.visitEntity(entity);
        this.parentEntities.push(entity);
    }

    @Override
    public void endVisitEntity(Entity entity) {
        this.parentEntities.pop();
        super.endVisitEntity(entity);
    }
}
```

Source Code 7.3: Code required for ParentAwareVisitor implementation. It emphasizes the excellent and straightforward implementation of custom operations.

To conclude, we were able to create our underlaying data-structure, including various rudimental functionality (database adaptation, REST interface, etc.), at velocity using the well-known generator, i.e., JHipster. However, the generated data-structure was not as general as it could be for what reason we added a hand-coded abstraction (`Entity` interface cf. Figure 7.3). With this abstraction, we were able to use the *Visitor-Pattern* to achieve an easy implementation of operations on our data-structure without altering the generated code further. The created `EntityVisitor` interface and furthermore its abstract implementation realized by the `BasicEntityVisitor` enable a very simple and straightforward realization of further operations, i.e., we only needed to implement four lines of functional code to realize our vital `ParentAwareVisitor`.

The resulting generic visit- and traversable traversable data-structure, as well as the three visitors, build the fundamental functionality for further implementation. Synthesizing this, we present the implementation of our two functional key concepts in the following, i.e., *Test Genericity* and *Whitelisting*.

7.2.3. Test Genericity

We realized the concept of *Test Genericity* by `Placeholder`, an entity representing a variable-like object inside the test-code, and a `PlaceholderValue`, an entity defining the value for instantiation. Next, we describe our approach to implementing the value-resolving-process (further called "placeholder resolving"), i.e., replacing all placeholder with their appropriate value. Therefore this process relies on a set of already derived placeholder values, given though `PlaceholderDerivation` (cf. Figure 7.5). The *derivation* process of `PlaceholderValues` is part of the *Whitelisting* concept. Since we describe this approach afterward, we postpone the detailed explanation of the `PlaceholderDerivation` and take it for granted.

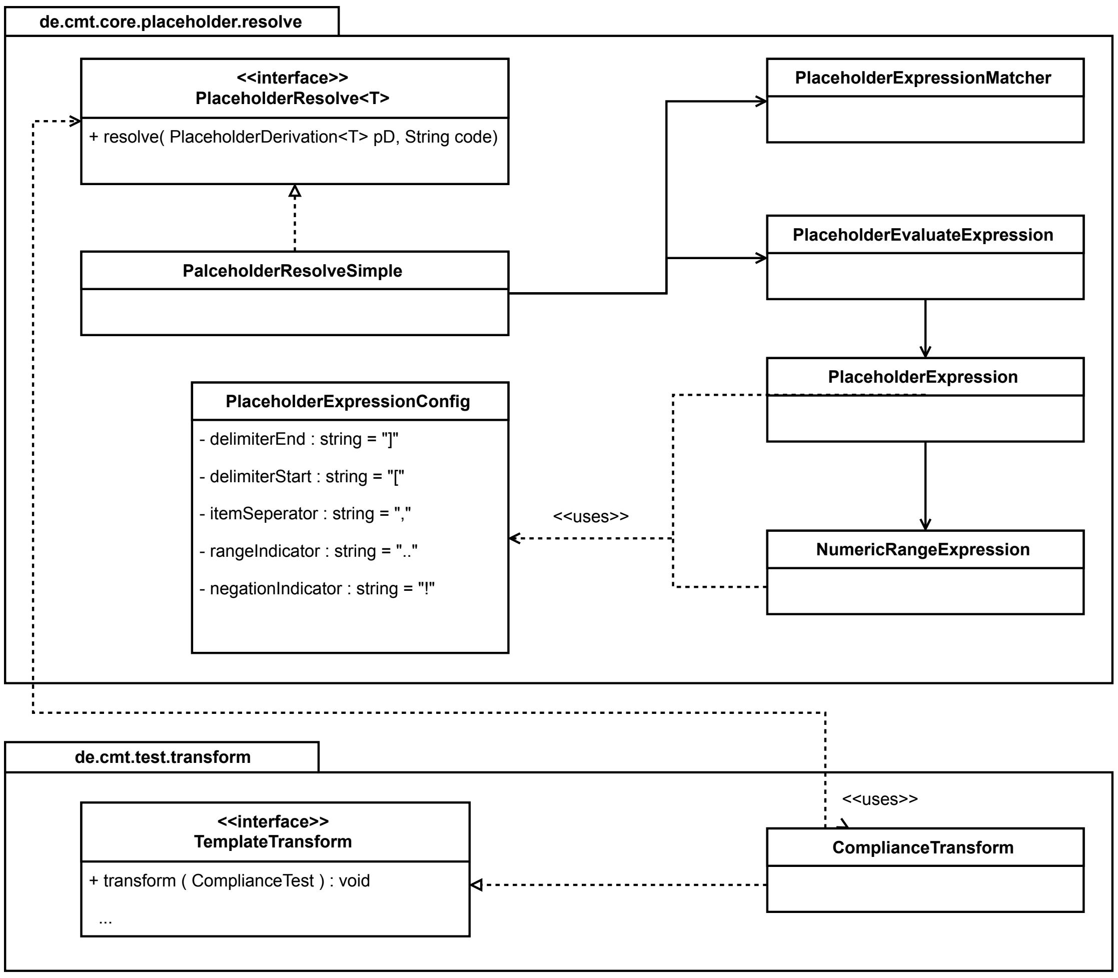

de.cmt.core.placeholder.resolve
<<interface>>
PlaceholderResolve<T>
+ resolve(PlaceholderDerivation<T> pD, String code)
PalceholderResolveSimple
PlaceholderExpressionMatcher
PlaceholderEvaluateExpression
PlaceholderExpression
PlaceholderExpressionConfig
- delimiterEnd : string = "]"
- delimiterStart : string = "["
- itemSeperator : string = ","
- rangeIndicator : string = ".."
- negationIndicator : string = "!"
<<uses>>
NumericRangeExpression
de.cmt.test.transform
<<uses>>
<<interface>>
TemplateTransform
+ transform (ComplianceTest) : void
...
ComplianceTransform

Figure 7.5.: UML class diagram illustrating the two main *packages* and their components which realize the code/template transformation functionality, i.e., placeholder expression resolving (top) and template transformation (bottom)

Placeholder resolving is applied during test code generation and transformation process (*Transform* component, cf. Section 7.1) to compile a "placeholder free" test code. Since these functionalities, i.e., a *transformation* of code and *resolving* of placeholder values, are independent of each other they should be coupled as loosely as possible. Thus we decided to apply to the *Strategy-Pattern*. This pattern allows us to define one or a set of algorithms, make them interchangeable and vary independently from the client that use it [Gam+95]. To this end, we created a generic interface `PlaceholderResolve` which itself acts as *Strategy*. We added genericity to this interface (`<T>`) to facilitate an appropriate adaptation of this functionality suitable to its context of usage. For example, one could be interested in resolving the placeholder value to string values (instantiation of `T` with `string`) or in resolving the object, i.e., instantiation of `T` with `PlaceholderValue`. Further, to fully apply the pattern, we defined a concrete strategy with `PlaceholderResolveSimple`. The context for `PlacholderResolve` primary is given through the `ComplianceTransform` located in `de.cmt.test.transform` (cf. Figure 7.5).

`PlaceholderResolveSimple` implements the `PlaceholderResolve` interface and instantiates the generic type `T` with `String`. It aims to resolve all included `Placeholder` to string values. In Section 6.4.1 we proposed to allow a *CMT-Exp* for `PlaceholderValues`, whereas this particular strategy needs the functionality to evaluate those. Because of this, the `PlaceholderResolveSimple` class is associated to a `PlaceholderExpressionMatcher` and `PlaceholderEvaluateExpression`. The former approach allows to extract all existing *CMT-Exps* from a given input string, and the latter instantiates all values for each matched expression, i.e., deriving a list of strings.

```
@Override
public void visit(ComplianceTest test) {
    super.visit(test);
    this.transform(test);
}
```

Source Code 7.4: Extending the *ParentAwareVisitor* visit function with an additional *transform* function.

This concept is realized by `ComplianceTransform` class, which acts as the strategy context and implements the *transformation* of test code.

`ComplianceTransform` extends the `ParentAwareVistor`, introduced in the previous section, to context-sensitively traverse the data structure. Each visit method, inherited through the `ParentAwareVistor`, is extended with an additional `transform` operation (cf. Listing 7.4). To support a plug-in based, CMT independent, transformation of each modeled entity the interface `TemplateTransform` is publicity offered as HotSpot. Thus a concrete implementation of the final transformation, e.g., file-based export or similar, is not done at this point. Nevertheless, as the plug-in should not derive and resolve the placeholder on its own, it is necessary to execute this beforehand. Thus the interface `transform(Test test)` implements the derivation and resolving process and alters the entity accordingly. In Listing 7.5 the conjunction of operations is shown.

Similar to the creation of the `ParentAwareVisitor`, we required only few code for this realization which justifies the previous abstraction and emphasizes the great adaptation of the *Visitor Pattern*.

```
@Override
public void transform(ComplianceTest test) {
    if( test.getPlaceholders().size() > 0 ) {

        HierachiePathUniqueDerivation pDerivation =
            new HierachiePathUniqueDerivation(
                this.getParentEntities()
        );
        test.setCode(
            this.placholderResolver.resolve( pDerivation,
                test.getCode() )
        );
    }
}
```

Source Code 7.5: Hooking into the *transform* interface function to alter the test code and replace all placeholder with appropriate values.

However, it should be mentioned that providing this *default* implementation the plug-in has to call the *super* method (`super.transform( entity )`) when implementing its specific transformation. Unfortunately, this could lead to misbehavior when used wrong.

To consolidate, we have implemented the proposed *Test Genericity* concept with the help of the *Strategy-Pattern*. Furthermore, we have outlined the creation of `PlaceholderResolveSimple` to implement placeholder *resolving* and the test code *transformation*, its plug-in interface and implementation as part of the `ComplianceTransform`.

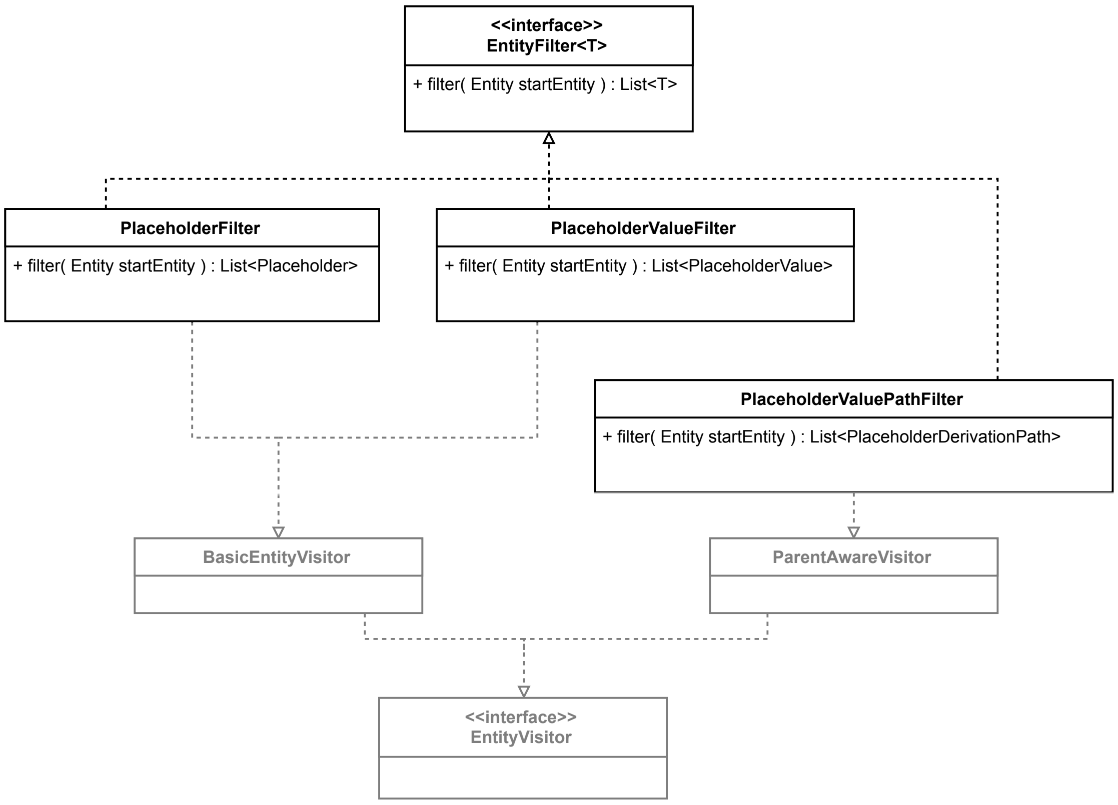
Next, we present the implementation of the suggested *Whitelisting* concept. This implementation includes the definition of the `PlaceholderDerivation` which was taken for granted in this section and utilized with `HierachiePathUniqueDerivation` in Listing 7.5.

7.2.4. Whitelisting

To realize the proposed concept of *Whitelisting* for placeholder value overriding, we implemented two separate but interconnected components.

First and more generally applicable, we implemented a *Filter* component using the *Strategy-Pattern*. This component builds upon on the previously introduced `Entity Visitor` to handle the object structure traversal itself. As illustrated in Figure 7.6 the filter strategy is realized with three concrete strategies, whereas each builds upon the `BasicEntityVisitor` or `ParentAwareVisitor`.

The added value of this component is the functionality to *filter* arbitrary elements with an given property out of our data model. It is comparable to the classic concept of any other filter. However, due to our *custom* data structure, we needed an own filter implementation and were not able to adapt existing once, e.g., `java.util.stream.filter`. To this end, we explain our custom implementations next.



<<interface>>
EntityFilter<T>
+ filter(Entity startEntity) : List<T>
PlaceholderFilter
+ filter(Entity startEntity) : List<Placeholder>
PlaceholderValueFilter
+ filter(Entity startEntity) : List<PlaceholderValue>
PlaceholderValuePathFilter
+ filter(Entity startEntity) : List<PlaceholderDerivationPath>
BasicEntityVisitor
ParentAwareVisitor
<<interface>>
EntityVisitor

Figure 7.6.: UML class diagram illustrating one adaptation of the *Strategy* pattern to realize the *filter* functionality. Slightly shown, it relies on the *EntityVisitor* to be able to operate on the crafted data structure.

PlaceholderFilter traverses the complete object structure and collects all `Placeholder`. When finished it returns a list of all linked `Placeholder`. This filter is primary used as part of the test simulation (cf. Section 7.2.5).

PlaceholderValueFilter similar to `PlaceholderFilter` this filter collects and returns a list of all defined `PlaceholderValues`. Likewise, it is used for the test simulation (cf. Section 7.2.5).

PlaceholderValuePathFilter enfolds the most complex filter implementation. Its main task is to filter all `PlaceholderValue` out of the current object structure. Furthermore, it groups those into sets which are identifiable through their context, i.e., the derivation path complied by all parent entities. As result one receives a list of `PlaceholderDerivationPath`, one for each derivated test. Each `PlaceholderDerivationPath` encapsulates the `Test`, the unique path identifier and all `PlaceholderValues` defined along the path from the root, e.g., `CustomerProject`, to a leaf, e.g., `Test` (cf. Listing 7.6). For example, this list would contain three different entries when deriving all `PlaceholderValue` for the test *CT3* in the exemplary scenario Figure 6.8 (cf. concept on Whitelisting, Section 6.4.1). This definition is necessary since a single `PlaceholderValue` can be part of various test derivations and therefore "exist" in different contexts.

This filter utilizes the `HierachieChainedDerivation` which is applied on every test visit (`visit(ComplianceTest test)`). It further filters all hierarchical inherited placeholder values, i.e., checking the correct derivation path as the `HierachieChainedDerivation` is context-insensitive (described next). Even more, it maps all `PlaceholderValue` to a new `PlaceholderDerivationPath`, whereas for each context (derivation path) a new instance is created. Storing all these into a list forms the outcome of this filter.

```
public class PlaceholderDerivationPath {

    private ComplianceTest      complianceTest;
    private String              pathIdentifier = "";
    private List<PlaceholderToValueMapping>
        derivedPlaceholderValues;

    ...
}
```

Source Code 7.6: PlaceholderDerivationPath
class which defines a list of PlaceholderValues for each, uniquely, derivable test case.

The second component implemented to realize this concept is defined by *Derivation*. Its core functionality is to provide a mapping from a `Placeholder` to one or a list of `PlaceholderValue`. This functionality is essential and used by others as *lookup* function to *derivate* placeholder. For example, this component was already used various time in previously presented implementation, e.g., `ComplianceTransform` or `PlaceholderValuePathFilter`

The usage of this mapping can vary due to different *view-points* or *interpretations* for what reason we decided to implement the *Strategy-Pattern* once again. Thus, the strategy interface (`PlaceholderDerivation`, cf. Figure 7.7) is always the same, but the final implementation or algorithm depends on the concrete strategy which should be adopted. We came up with three different concrete strategies which are used in various contexts likewise.

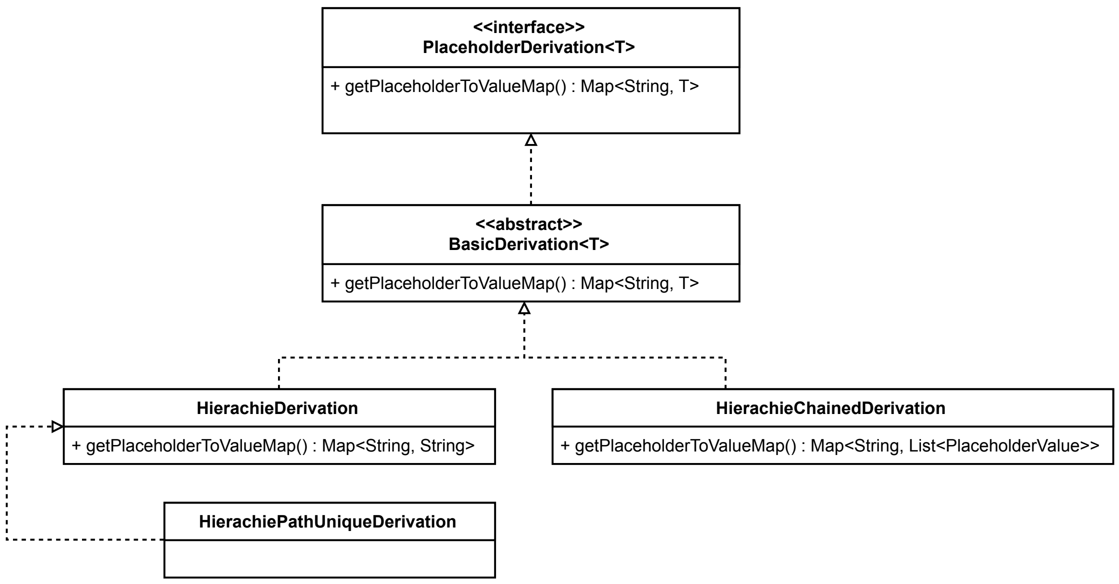

<<interface>>
PlaceholderDerivation<T>
+ getPlaceholderToValueMap() : Map<String, T>
<<abstract>>
BasicDerivation<T>
+ getPlaceholderToValueMap() : Map<String, T>
HierachieDerivation
+ getPlaceholderToValueMap() : Map<String, String>
HierachieChainedDerivation
+ getPlaceholderToValueMap() : Map<String, List<PlaceholderValue>>
HierachiePathUniqueDerivation

Figure 7.7.: UML class digramm illustrating the major classes and interfaces used to implement the *placeholder derivation* functionality. The *Strategy* pattern is adapted to allow different *strategies*, i.e., *HierachieDerivation*, in varying contexts.

HierachieDerivation implements the most simple mapping and derivation strategy, more precisely the mapping from a placeholder key, given as a string, to a placeholder value, likewise resolved as a string. The derivation is *context-insensitive*, whereas the highest (superior) entity overrides all child definition. For example, a placeholder value definition on `CustomerProject` level would override/set the value for all derived tests, independent from its context.

HierachieChainedDerivation operates in the same way as the `HierachieDerivation`, i.e., *context-insensitive*, but additional stores all previously defined placeholder values in a list (*chained*). Thus, the mapping yields a list of placeholder values for each placeholder, whereas the uppermost entity represents the overwriting-winning entity.

HierachiePathUniqueDerivation extends the `HierachieDerivation` and thus yields a string-to-string mapping as well. However, this filter is, in contrast to its superclass, *context-sensitive* and allows an overriding definition for one or more unique derivation paths. To realize this it utilizes and compares the `pathIdentifier` of the `PlacehoderValue`. For example, with the help of this derivation strategy, one could define a path-unique `PlacehoderValue` for context A (leftmost path) in the scenario shown in Figure 6.8.

To this end,the `HierachiePathUniqueDerivation` represents a derivation technique which allows us to implement and realize the proposed concept of *Whitelisting*. This functionality justifies the adaptation and usage of the `HierachiePathUnique Derivation` in previous outlined implementation, i.e., test code generation. The other implemented and presented derivation strategies, i.e., `HierachieDerivation` and `HierachieChainedDerivation`, are primarily used for simulation.

7.2.5. Compliance Testing

The compliance testing, e.g., execution or simulation of modeled compliance tests, is accomplished through the *Adaptation* component as explained in the architecture blueprint in Section 7.1. We distinguished between the disjoint activities *execution* and *simulation* and created an individual component each. Both actions are accessible through the *TestResource* and, more precisely, used by the `ComplianceRun` as illustrated in Figure 7.8 and Figure 7.10. We separately describe their implementation in the following.

Execution

The final test execution is achieved through on or more *Test Plug-ins*, which are fairly independent from the remaining backend. For an exemplary implementation, we realized the adaptation of *InSpec*. All its technologies specific code is located in the package `de.cmt.test.plugin.inspec`.

To facilitate an adaptation as `TestRunner` inside the CMT control flow, the plug-in has to implement the provided `TestRunner` interface (cf. Figure 7.8). The overall execution process is prescribed as a set of distinct, chronological ordered task, i.e., test code generation, test execution and result evaluation. Each of these tasks needs to be realized by the plug-in itself. Thus we present the InSpec specific implementation of each task in the following.

Test Code Generation is implemented by `InspecTransform` which extends the abstract class `ComplianceTransform`. To create the InSpec specific file format `InspecTransform` uses the *Freemarker* template engine provided by its superclass. Furthermore, it implements all `transform()` interfaces required by `TemplateTransform` (cf. Figure 7.5). Each transformation method aggregates certain data of the entity transformation and afterward applies the `createProfile` or creates the specific itself.

Placeholder derivation which is required for concrete test instantiation is indirectly done through the abstract class `ComplianceTransform`, thus the plug-in does not need to handle the derivation by its own.

Test Execution is realized via `InspecCommand`. As the naming indicates this class allows an command creation which is later executed in the system shell. To receive an easy and system-independent command execution we implemented `InspecCommmand` on the basis of the `CommandLine` class provided by the *Apache Common Execution* library. Furthermore we implemented an extension of the default command, namely `InspecDockerCommand`, which allows the usage of InSpec inside a Docker container.

Result Evaluation is carried out by the `InspecEvaluate` class. Its only function is to parse and convert the InSpec specific created test results, i.e., json format, into a *CMT* conform format, e.g., into a `JobResult`. Furthermore, the overall result, representing the test failure or succeeding, is calculated and stored.

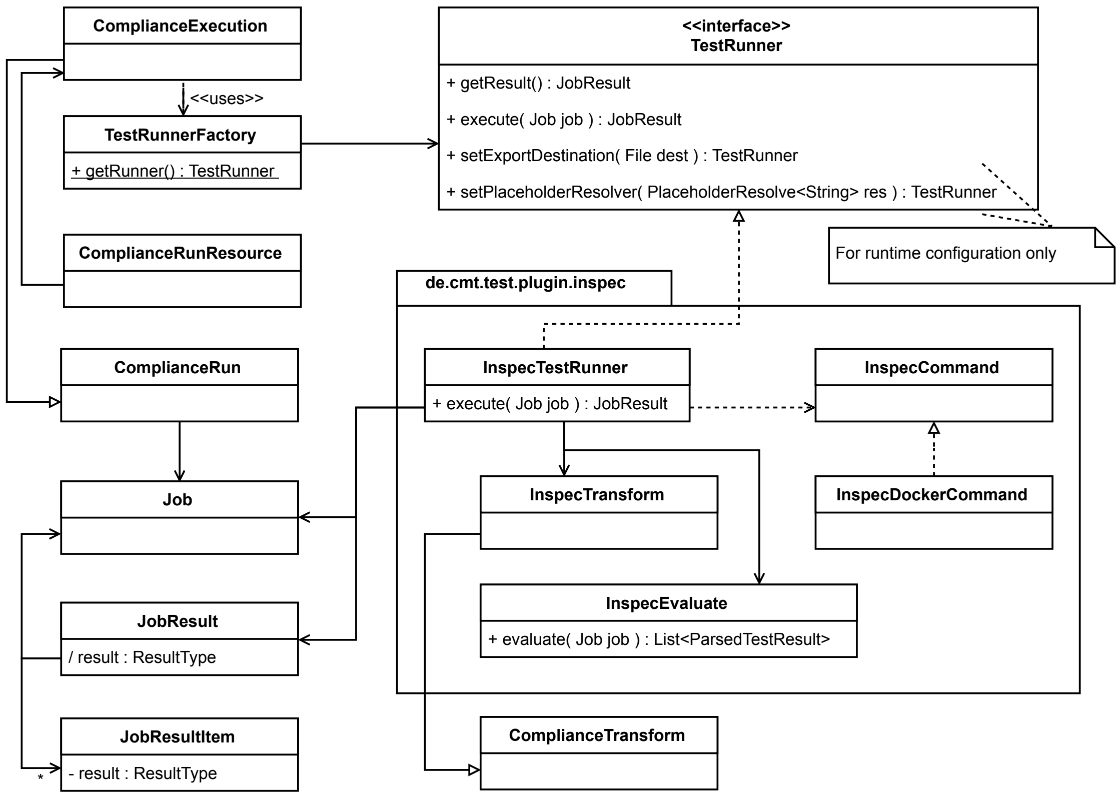
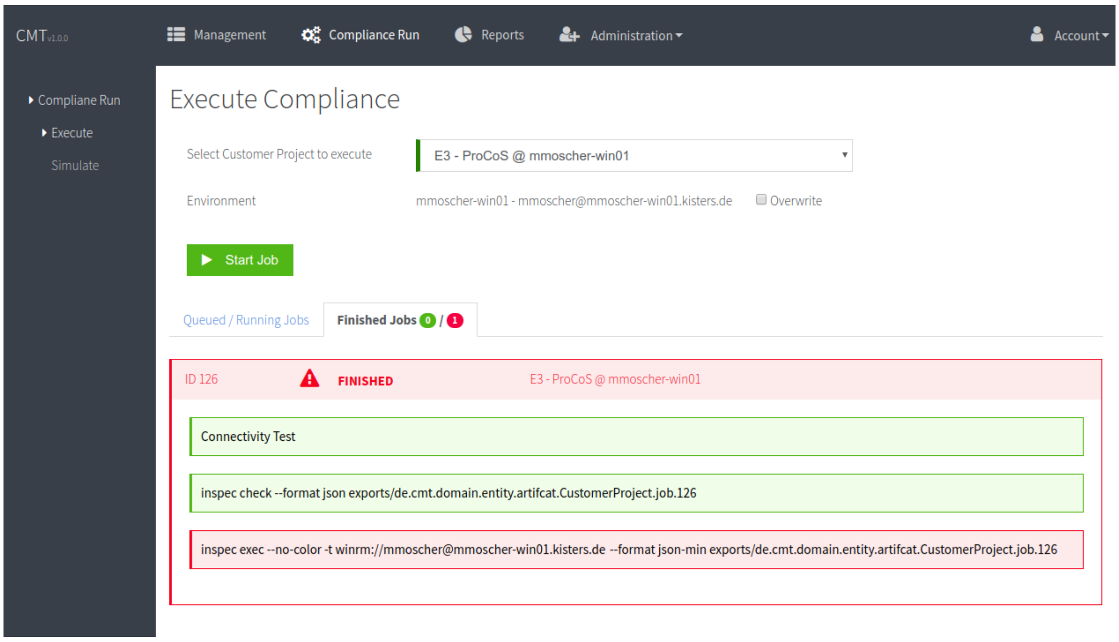

ComplianceExecution
<<uses>>
TestRunnerFactory
+ getRunner() : TestRunner
ComplianceRunResource
ComplianceRun
Job
JobResult
/ result : ResultType
JobResultItem
- result : ResultType
*
<<interface>>
TestRunner
+ getResult() : JobResult
+ execute(Job job) : JobResult
+ setExportDestination(File dest) : TestRunner
+ setPlaceholderResolver(PlaceholderResolve<String> res) : TestRunner
For runtime configuration only
de.cmt.test.plugin.inspec
InspecTestRunner
+ execute(Job job) : JobResult
InspecCommand
InspecTransform
InspecDockerCommand
InspecEvaluate
+ evaluate(Job job) : List<ParsedTestResult>
ComplianceTransform

Figure 7.8.: UML class digram illustrating the major *classes* and *interfaces* used to implement the *InSpecTestRunner*. The main functionality is encapsulated in the *de.cmt.plugin.inspec* package. To allow external access (contracts) framework specific interfaces are implemented, e.g.,*TestRunenr*

The orchestration of these tasks is done by the central, and publicity known, class `InspecTestRunner`. This class acts as "single-point-of-interest" for the CMT Framework and implements the `TestRunner` interface to support external accessibility. For the sake of simplicity the InSpec plug-in, more precisely the `InspecTestRunner`, yet relies on some domain classes, i.e., `Job`, `JobResult`, and `JobResultItem`, to access required data during test generation and to store the derived results in a framework conform format. However, this could be implemented in an more abstract manner in further versions. For the realization of an *proof-of-concept* implementation we admit this slightly increased coupling.



Select Customer Project to execute: E3 - ProCoS @ mmoscher-win01

Environment: mmoscher-win01 - mmoscher@mmoscher-win01.kisters.de ☐ Overwrite

ID 126 FINISHED E3 - ProCoS @ mmoscher-win01

```
Connectivity Test
inspec check --format json exports/de.cmt.domain.entity.artifcat.CustomerProject.job.126
inspec exec --no-color -t winrm://mmoscher@mmoscher-win01.kisters.de --format json-min exports/de.cmt.domain.entity.artifcat.CustomerProject.job.126
```

Figure 7.9.: Exemplary screenshot of the *Execute Compliance* web interface. It illustrates one failed execution of the *CustomerProject* E3 - ProCoS.

Simulation

Beside test execution, we proposed the realization of a *simulation*. This should allow to analyze, validate or visualize the modeled compliance requirements. Each of those actions is achieved with a specific implementation implementing the `ValidationStrategy`. To operate in a universal way, the `ValidationStrategy` requires a `ValidationContext` as input on which it performs its action. The `ValidationContext` comprises a *context* given through an `ArtifactEntity` and a set of *validationResults*, one for each executed `ValidationStrategy` since the same `ValidationContext` can be used in other strategies as well. To be able to traverse and filter the passed context, e.g., the `ArtifactEntity`, the `ValidationStrategy` utilizes an `EntityFilter`.

Exemplary, we have realized to different validation strategies which are applied as part of the `ComplianceSimulation`. The retrieved results are stored in a `JobResult` entity, similar to the *execution* realization.

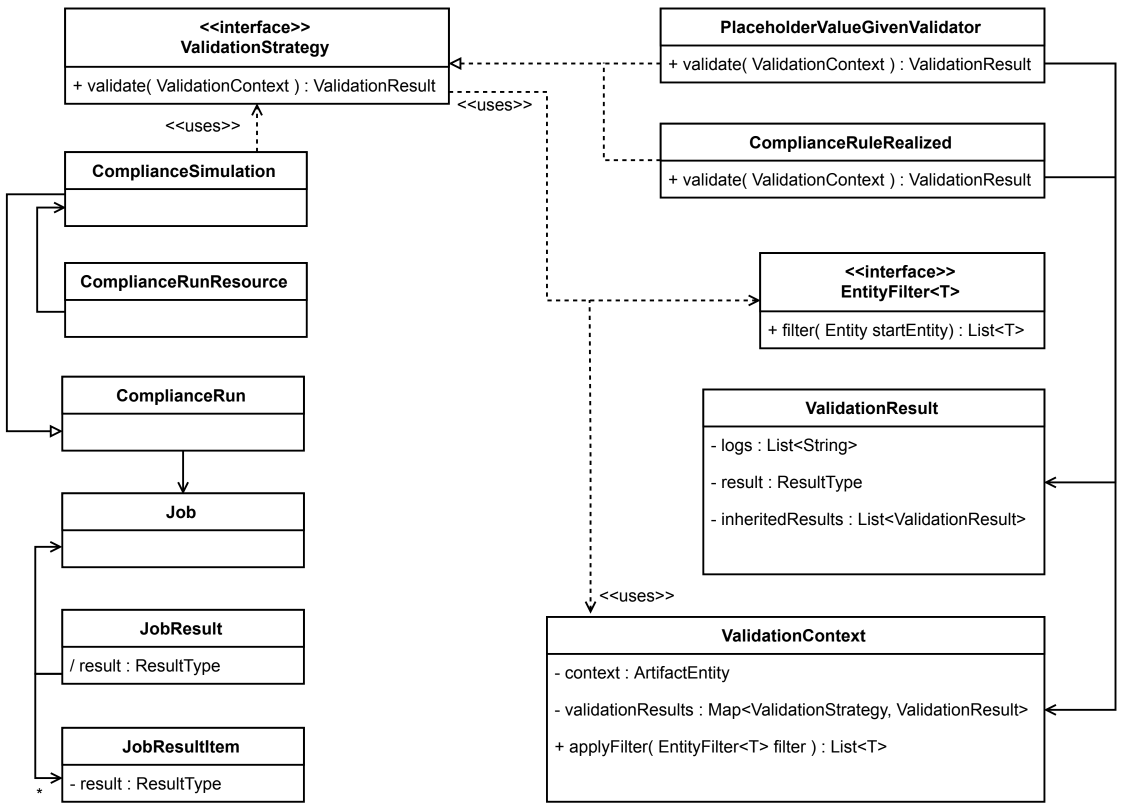

<<interface>>
ValidationStrategy
+ validate(ValidationContext) : ValidationResult
<<uses>>
ComplianceSimulation
ComplianceRunResource
ComplianceRun
Job
JobResult
/ result : ResultType
JobResultItem
- result : ResultType
*
PlaceholderValueGivenValidator
+ validate(ValidationContext) : ValidationResult
<<uses>>
ComplianceRuleRealized
+ validate(ValidationContext) : ValidationResult
<<interface>>
EntityFilter<T>
+ filter(Entity startEntity) : List<T>
ValidationResult
- logs : List<String>
- result : ResultType
- inheritedResults : List<ValidationResult>
<<uses>>
ValidationContext
- context : ArtifactEntity
- validationResults : Map<ValidationStrategy, ValidationResult>
+ applyFilter(EntityFilter<T> filter) : List<T>

Figure 7.10.: UML class diagram illustrating the realization of compliance test *simulation*. As shown, it relies on the *EntityFilter* and defines additional *ValidationStrategies*. During each *ComplianceSimulation* one or multiple *ValidationStrategies* are used to *validate* the *ValidationContext*.

PlaceholderValueDefinedValidator first filters all *Placeholder* and *PlaceholderValues* attached to the context (*ArtifactEntity*) using the `applyFilter()` method (cf. Listing 7.7). With the aid of these sets, it validates if each used *Placeholder* has at least on value definition.

This validation is useed to check if the modeled context is complete and thus executable.

ComplianceRuleRealizedValidator can be applied to check if every derivable compliance rule in the current context has at least one test. Therefore this validator applies the `ComplianceRuleFilter` to extract all compliance rules and further checks if one or more tests are linked. If a compliance rule with less than one test is found, this validation fails.

Similar to the `PlaceholderValueDefinedValidator` is useed to check the practicability of the current context.

```

@Override
public ValidationResult validate( ValidationContext context ) {

    List<Placeholder> placeholder =
        context.applyFilter( new PlaceholderFilter() );
    List<PlaceholderValue> placeholderValues =
        context.applyFilter( new PlaceholderValueFilter() );

    ...
}
```

Source Code 7.7: Filtering *Placeholder* and *PlaceholderValue* on an arbitrary context as part of the PlaceholderValueDefinedValidator

However, we can imagine that further interesting validators or analyzer exists and could be realized. To extend the overall *ComplianceSimulation* with an additional *ValidationStrategy* or exchange the existing once, one can easily alter the existing setup compiled in Listing 7.8.

```

@Override
public synchronized void setExecutionJob() {

    this.validationContext.setContext( this.job.getArtifact() );

    this.validations.add( new PlaceholderValueDefinedValidator() );
    this.validations.add( new ComplianceRuleRealizedValidator() );
}
```

Source Code 7.8: *ComplianceSimulation* setup, i.e., registering of all applicable validations strategies.

7.2.6. Summary

In this section, we gave a rough overview of our implementations and the main functionalities which realize the core concepts of the *Compliance Management Tool*.

First, we have shortly outlined the basic technology stack, i.e, Spring/Java, and the generator used to create the fundamental application at high velocity.

In Section 7.2.2, we described the creation and further generalization of the domain model in much detail. We argued the application of the *Visitor-Pattern* which realizes the overall bedrock functionality to operate on our data model in various way. For example, we introduced the `EntityVistor`, `EntityFilter` and several manifestation of these, e.g., the `ParentAwareVisitor` which enables traversing the data model in a context-sensitive way.

Based on the *Compliance Model* realization, we accurately described the concrete implementation of the two functional concepts *Test Genericity* and *Whitelisting*. Their adaptation was explained as part of the realization of the *Compliance Testing* in Section 7.2.5.

To conclude, we have managed to create a very modular realization which can be extended at various points through the implementation of further concrete strategies. Furthermore, we accomplished to introduce a great, additional, layer of abstraction on the generated domain model. On the basis of this, the yet realized domain model (Compliance and Artifact entities) is easily extendable and still usable due to the generic implemented visitors.

7.3. Execution Tooling

To accomplish a very straightforward integration into automation systems like Jenkins, we presented the concept of an additional *Command Line Interface* which should act as a non-graphical frontend for the *Compliance Management Tooling*. As the communication with the CMT is done with the REST paradigm, we decided to use a scripting language which comes with REST communication support. To this end, we chose *Python* as the scripting language to realizes our CLI for the CMT backend.

The rough class and inheritance structure, if one can classify this as such, is posed in Figure 7.11.

The central functionality is the simplification of communication effort, wherefore each class extends the *CMT_Conncetion* class to accomplish the required HTTP request. As the class naming indicates, the CLI provides four major functionality, i.e., a listing of various entities (*CMT_Listing*), presenting a result of finished jobs (*CMT_Result*), kicking off new jobs for an specific artifact (*CMT_Runner*) and importing baseline test written in a cmt-cli specific format (*CMT_Import*). All these functionalities are accumulated in the *CMT_Client* command class, which delegates the user input (via command line) to the specific function handlers. Accordingly, to the available actions the *CMT_Client* requires one – out of four – positional arguments (*run, result, list or import*) to execute the desired task. Listing 7.9 shows the compiled help documentation.

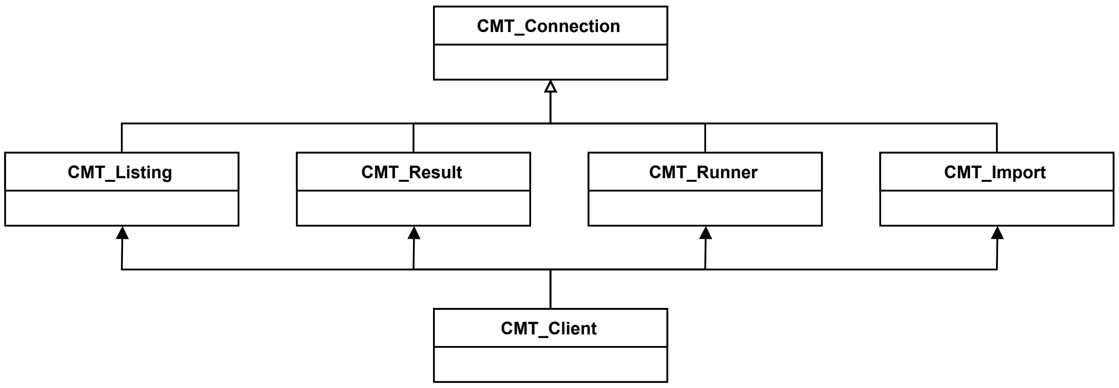


Figure 7.11.: UML class diagram illustrating the six main classes used to realize the *CMT-CLI*. As illustrated, all classes extend the *CMT_Connection* to implement the communication protocol.

```
[root@mmoscher-dev ~]# ./cmt.py --help

usage: cmt.py [-h] [-u USERNAME] [-p PASSWORD] [-H HOST]
              {run,result,list,import} ...

Command Line Interface to access a Compliance Management Tooling
    (CMT) backend

positional arguments:
    {run,result,list,import}
        run             command to start a compliance run
        result          command to receive results of a job
        list            command to list a certain resources
        import          command to import CMT Testfiles

optional arguments:
    -h, --help
        show this help message and exit
    -u USERNAME, --user USERNAME
        username for login, default: admin
    -p PASSWORD, --password PASSWORD
        password for login, default: admin
    -H HOST, --host HOST
        Host address for backend, default: http://localhost:8080
```

Source Code 7.9: Compiled help documentation for the cmt-cli python script.

Since this script only acts as wrapping function to the already introduced and explained CMT functionality, we intentionally skip further detailed description of these. However, the *import* operations requires `Tests` in a well-defined format, i.e., the *cmt-cli* file format based on YAML. Such an exemplary test, defined using the YAML (yml) format, is compiled in Listing 7.10. The four major and required keywords are *name*, *placeholder*, *description*, and *code*.

```
name: E3 - TCP Port Listening

placeholder:
    - key: port
      description: single or range of ports to be tested

description: |
    Check if a service (tcp connection) is listening on certain
       port

code: |
    describe port(<% tcpPort %>) do

    it { should be_listening }
        its('protocols') { should include 'tcp' }
    end
```

Source Code 7.10: Exemplary port test defined using the yml format to allow an import via the CMT CLI.

Summarizing it is to say that we have successfully created a command-line-interface script which allows easy remote control of the CMT. Furthermore, we presented a possibility to easily import predefined tests.

7.4. Reporting

Primary the *Evaluate* process phases comprise certain management decision-making, e.g., deciding on actions how to handle a compliance test failure for a product. Since these decisions highly depend on the vendors compliance violation handling strategies, and as we do not possess the require expert knowledge on this domain, we can not outline (realize) a guideline on how to handle test failures. However, as proposed in the related concept, we can undertake some activities to support this process as much as possible. To this end, we have implemented the conceptual presented KPIs, i.e., *Maximum Compliance Violation* and *Compliance Index.* To access those metrics, we have created an additional reporting screen in our frontend component. The concrete implementation is realized in the backend component and is illustrated in Figure 7.12.

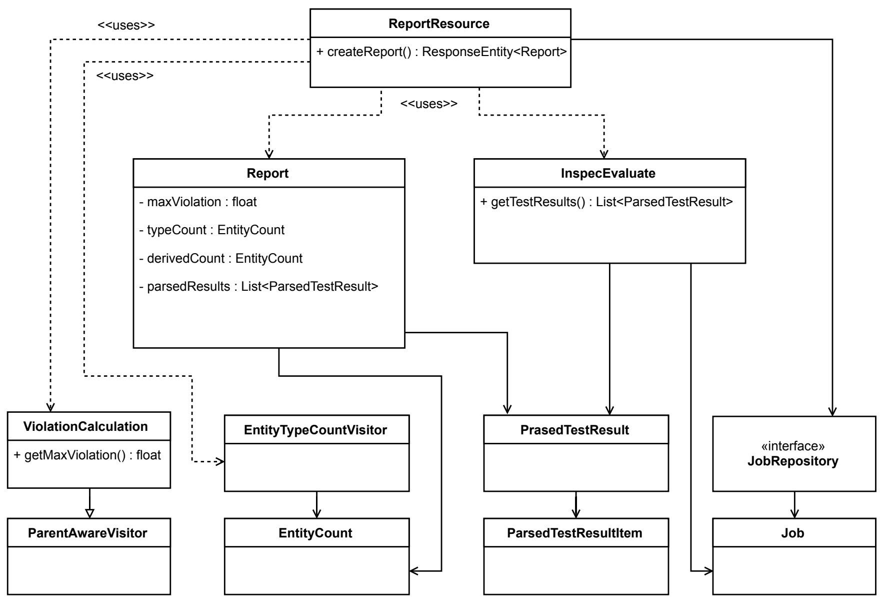


Figure 7.12.: UML class diagram illustrating the central classes used to realize the *Reporting* concept.

The access to the reporting functionality and its derived data, i.e., *Report*, is given through the `ReportResource` component located in the `de.cmt.communication.reporting` package. Its main task is to fetch and evaluate the test results for a given *job-id* and further perform different actions on these. The received results are than stored in a *Report* instance and passed back to the client. The wrapping class *Report* encapsulates certain information of interest for the reporting process, i.e., *maxViolation*, *typeCount*, *derivedCount* and *parsedResult*. Their detailed creation and the further utilized classes (cf. Figure 7.12) are explained in the following. An overview of all taken operations and the creation of a *Report* is compiled in Listing 7.11.

```
 1  ViolationCalculation vClac = new ViolationCalculation();
 2  Report report = new Report();
 3
 4  Job job = jobRepository.findOne(id);
 5  CustomerProject project = (CustomerProject) job.getArtifact();
 6  InspecEvaluate jsonEval = new InspecEvaluate( .. );
 7
 8  EntityTypeCountVisitor uniqueCount = new EntityTypeCountVisitor();
 9  EntityTypeCountVisitor derviedCount = new
       EntityTypeCountVisitor(false);
10  uniqueCount.handle( project );
11  derviedCount.handle( project );
12
13  report.setTestsViolation( vClac.getTestsViolation(project));
14  report.setDerviedCount( derviedCount.getEntityTypeCount() );
15  report.setTypeCount( uniqueCount.getEntityTypeCount() );
16  report.setParsedResults( jsonEval.getTestResults() );
17  report.setMaxViolation( vClac.getMaxViolation() );
```

Source Code 7.11: Creation of a report for a given job-id

In the very beginning, the `InspecEvaluate` implementation (described in Section 7.2.5) is used to extract the test result for every single test, i.e., if it has passed or not (*line 13*). Since this interpretation is plug-in specific, we do rely on its concrete implementation. The evaluated results are encapsulated in a `ParsedTestResult` instance, which itself contains further `PrasedTestResultItems` - for each test one item.

Next the `EntityTypeCountVisitor` is applied (*line 14-15*). This class is an implementation of the `BasicEntityVisitor` (cf. Section 7.2.2) and simply counts all entity occurrences. Depending on the initial flag passed to its constructor the visitor counts every occurrences (amount/indication of entity reuse) of an entity type or the amount of entity per type used in the data model (amount of unique entities, indented via its id). The derived results are passed as an `EntityCount` instance.

Finally, the class `ViolationCalculation` is used to calculate the *Maximum Compliance Violation* as well as the specific violation score for each test (*line 16-17*). Since it needs to traverse the entire data model, e.g., to derive all necessary information as defined in the calculation concept (cf. Section 6.4.3) it extends the *ParentAwareVisitor*.

8. Evaluation

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

AUTHOR UNKNOWN

Contents

To evaluate our presented process model *Continuous Compliance* and its adaptation in the form of *Continuous Compliance Testing*, we conduct four different aspects.

First and foremost we assess quality of the *Compliance Management Tooling (CMT)*, i.e., its validity. Therefore, we *verify* the previously defined software requirements Section 8.1 and furthermore *validate* its support of *Continuous Compliance* in the context of *ProCoS* by applying a case study, which describe it detailed in Section 8.2.

Next, referring to the obtained compliance models (*Maximum Reuse*, *Naive* and *Balanced*) for ProCoS, we evaluate the *intra-model* reusability and discuss further possible reusability capabilities, i.e., *inter-model.*

Lastly, in Section 8.4, we review and describe the integration and support of other process models. We want to verify additional value added by our *Compliance Management Tooling* beyond our modest process model.

8.1. Verification

To assure if the created *Compliance Management Tooling* fully satisfies all formulated and expected requirements (cf. Section 6.2), we perform a rough *software verification* [Poh10], i.e., we review the created software components (cf. Section 7.1) against the defined requirements.

The domain related requirements, e.g., modeling and linking certain entities, defined by requirement *[CMR-Req-1]*, *[CMR-Req-2]*, *[CMR-Req-3]*, *[CMR-Req-4]*, *[CMR-Req-6]*, and *[CMR-Req-8]*, are realized by the most centric software component *Domain*. Its accurate implementation was explicitly explained by the *Compliance Model* realization in Section 7.2.2. To support a *Placeholder* in *Test Code* to yield *Test Genericity* (cf. *[CMR-Req-5]*) the *Core* component includes the particular software component *Placeholder*. Its realization, e.g., *Placeholder Determination* and *Derivation*, was implemented and discussed in Section 7.2.3. For a practical test execution and simulation, i.e., compliance adaptation demand by *[CMR-Req-7]*, the *CMT* provides an *Adaptation* and *Test* software component. Finally, the last requirement *[CMR-Req-9]*, i.e., support of result reports, is realized by the *Reporting* components. Its accurate implementation was described in Section 7.4.

To summarize, we successfully verified that the implemented blackbox framework [FS97], i.e., the *Compliance Management Tooling*, realizes all described software requirements.

8.2. Validation – Case Study KISTERS ProCoS

To validate the desired tool-support during *Continuous Compliance* we conduct a case study which is precisely classified next.

Case Study Classification For the validation of the *Compliance Management Tooling* and its support of the presented *Continuous Compliance* process we implement a *exploratory* case study, i.e., finding out what is happening, seeking new insights, and generating ideas and hypotheses for new research [RH08], based on *qualitative* data, i.e., data which involves words, descriptions, pictures, and diagrams [RH08]. To this end, we explicitly define the *qualitative* dataset and the *objective*, i.e, a statement of what is expected to be achieved in this case study.

Qualitative Dataset To accomplish the case study we selected *ProCoS* provided our cooperation partner KISTERS. *ProCoS* a process control system and is suited for the application in various domains, e.g., Energy- and Water- supply or infrastructure application.

Objective The central objective of this case study is to evaluate the adaptation and usability of the created tool support for the *Continuous Compliance Testing*. It should be clarified *how* and *to which extend* the tooling is able to facilitate a practical adaptation of *Continuous Compliance*.

Next, to perform the previously defined case study, we apply all three phases of *Continuous Compliance* using the *CMT* on the chose *Qualitative Dataset*. However, due to to personal restriction and nonexistent expert knowledge, we deliberately drop the detailed evaluation of each noted responsibility of involved roles.

First, accordingly to the phase *Elaborate*, we analysis and compile a set of different compliance and baseline requirements (cf. Section 8.2.1). Next, in Section 8.2.2, we shortly describe how to use and create a *Test* to verify an exemplary ProCoS requirement, i.e., *ProCoS_Install #2*. On this basis, we present and discuss three different attempts taken to model all previous complied ProCoS compliance requirements. The reporting on the received results, i.e., executing the *Balanced* model on out ProCoS test environment, is shortly discussed in Section 8.2.3.

We split each phase evaluation into three fine-grained sections, i.e., *Application*, *Discussion* and an *Interim Conclusion*. The latter one will facilitate the final conclusion outlined in Section 8.2.4.

8.2.1. Continuous Compliance – Elaborate

In this first phase of *Continuous Compliance*, i.e., *Elaborate*, we aim to analyze and define compliance requirements for ProCoS.

Application

The application of this phase is split into two distinct fields of requirements extraction. First and foremost we describe the derivation of ProCoS specific compliance requirements. Afterwards, we outline additional specification applicable as *baseline* compliance, i.e., Windows baseline tests.

Compliance Requirements To model the product specific compliance (and configuration) required by ProCoS, we conducted several technical documentation concerning the fundamental installation process and furthermore the operation in a virtual environment and its specific settings. We complied a list of all provided documents[1] in the following enumerations and reference each of them by the preceded abbreviation.

ProCoS_AntiVirus – `ProCoS_und_Antivirenprogramme.pdf`
Specific permission management when running ProCoS in conjunction with a vendor independent anti-virus software.

ProCoS_Dir – `handout-ProCoS-VERZEICHNIS.docx`
Listing of all relevant directories needed by ProCoS during Runtime.

[1]All documents were provided electronically on the June 23rd, 2017. Thus, the derived compliance rules rely on the outlined requirements, configurations and permissions for ProCoS from that very time.

ProCoS_Network – `Procos870_im_Netzwerk.pdf`
Required Network interface- and sharing configuration to successfully operate ProCoS in a network-based environment.

ProCoS_Install – `I-A.Installation_ProCoS-Windows7-2017-06.doc`
Comprehensive and very detailed installation manual for the "'ProCoS Leitsystem" on a Windows 7 operating system.

ProCoS_VM – `ProCoS_in_VM_Umgebung-201505.pdf`
Hard- and Softwarerequirements for ProCoS when running in a virtual environment, e.g. VMWare or VirtualBox.

In the following, we present a table for each document representing the extracted (relevant) information, which will be used for test case derivation later. As some documents cover both, server and client application of ProCoS, each row will indicate for which version (or both) the compliance demand is designated. If a requirement was already defined or mentioned previously, we will point to this definition. Furthermore, we deliberately go without *atomic Compliance*, because in our opinion this is a design decision (cf. Section 8.2.2) and should be handled as part of the(next) modeling phase.

Requirements based on ProCoS_AntiVirus

Req. No	Server	Client	Requirement
#1	✓	✓	Check if files and directories listed in the document (18 in total) are correctly configured[2] for exclusion in the appropriate Virus Scanner, i.e. Windows Defender.

Requirements based on ProCoS_Dir

Req. No	Server	Client	Requirement
#1	✓	✓	Check if the correct folder structure, including all sub-folder, exists and is read/writable by correct user (*ProCoS*)

[2]https://www.tenforums.com/tutorials/5924-add-remove-windows-defender-exclusions-windows-10-a.html

Requirements based on ProCoS_Network

Req. No	Server	Client	Requirement
#1	✓	✓	Check if both can reach the corresponding opponent via IPC network share without authentication (or other user interaction).
#2	✗	✓	Can access Firebird RDBMS on port TCP:3050 and TCP:3060
#3	✓	✗	(RDBMS) Ports TCP:3050 and TCP:3060 are open for external access
#4	✓	✓	Ports open and accessible **TCP: 137, 138, 139, 445** Windows file sharing (incl. NETBIOS) **TCP: 3050, 3060** Firebird RDBMS Service **TCP: 3389** Remote Desktop Protocol
#5	✓	✗	Windows Service *LanmanServer* (Server) and *RpcSs* (Remoteprocedurecall) installed and configured to autostart

Requirements based on ProCoS_Install For the ProCoS (basic) installation the more general compliance requirements regarding Network configuration remain valid. This means, we only outline additional or deviating demands. Furthermore, we omitted all *User Experience* (UX) related settings and configuration since those do not have – to the best of our knowledge and belief – any impact on security.

Req. No	Server	Client	Requirement
#1	✓	✓	Separate partition for the ProCoS installation exists and is formatted as NTFS
#2	✓	✓	Operating Systems partition (C:) needs to have at least 40GB
#3	✓	✓	Correct user accounts and permissions (Administrator, ProCoS, Admin) (cf. Document Section 3.2) are configured
#4	✓	✓	Correct group and permission (*ProCoS-Benutzer*) exist (cf. Document Section 3.2)
#5	✓	✓	Default user and groups are deactivated (cf. Document Section 3.2)

#6	✓	✓	Autologon[3] is installed
#7	✓	✓	Remote access is configured for group *Administrator*
#8	✓	✓	(hidden) Administrator account is activated [4]
#9	✓	✓	Windows Update is deactivated
#10	✓	✓	Windows Firewall is deactivated
#11	✓	✓	Ports open and accessible **TCP: 135, 139, 445** RPC, NetBIOS **TCP: 3050, 3060** Firebird RDBMS Service **TCP: 2404, 8000, 8100** IEC-10x **TCP: 80, 8080** HTTP/HTTPs **TCP: 443** SSL **TCP: 12345** ProCoS Apache **TCP: 5631** PcAnywhere **UDP: 137, 138, 445** RPC, NetBIOS **UDP: 5632** PcAnywhere
#12	✓	✓	Anti Virus is configured to exclude the folders *db*, *Dta* and *Def* located in the ProCoS installation directory.
#13	✓	✓	users group (*ProCoS-Benutzer*) has permission to modify system time
#14	✓	✓	System time synchronization through Internet is deactivated
#15	✓	✓	Network shares (naming and path) for user group (*ProCoS-Benutzer*) are setup (cf. Document Section 3.9)
#16	✓	✓	Hostname is composed of (A-Z,0-9,-) only
#17	✓	✓	*NetBEUI* protocol and *QoS* packet planner are deactivated
#18	✓	✓	*Client for Microsoft Networks* is not installed
#19	✓	✓	IPv6 is disabled
#20	✓	✓	NetBIOS is activated for TCP/IP
#21	✓	✗	Additional software (tools) are located in the *Tools* directory (cf. Document Section 4)

[3]https://technet.microsoft.com/de-de/sysinternals/autologon.aspx

[4]https://www.tenforums.com/tutorials/2969-enable-disable-elevated-administrator-account-windows-10-a.html

#22	✓	✗	A symbolic link to *Startup.cmd* is placed in *All Users\Startmenü\Programme\Autostart* (cf. Document Section 4.1)
#23	✗	✓	A symbolic link to *apupdate.cmd* is placed in *All Users\Startmenü\Programme\Autostart* (cf. Document Section 4.1)
#24	✓	✓	ProCoS OLE-Interface is installed and located in the ProCoS installation directory (cf. Document Section 4.2)
#25	✓	✗	Firebird (KISTERS Deployment) is installed (cf. Document Section 4.4)
#26	✓	✗	Firebird config (aliases.conf) contains ProCoS databases paths (cf. Document Section 4.4)
#27	✓	✗	Application *DatenNetz* and Script *hhdsich.cmd* are available to preform backup (cf. Document Section 4.5)

Requirements based on ProCoS_VM To run ProCoS inside a virtual machine (VM) the more general compliance requirements concerning AntiVirus and Network configuration remain valid. This means, we only outline additional or deviating demands.

Req. No	Server	Client	Requirement
#1	✓	✓	When using Windows-Autoupdate the automatic installation of (downloaded) updates needs to be deactivated and done manually.

As compiled above we were able to extract 35 requirements for ProCos either running in *server* or *client* mode.

Next we will shortly outline the windows baseline requirements which could be used in conjunction with the ProCoS specific compliance.

Windows Baseline Requirements We created a baseline test for Windows 10 since a further motivation (indirect outcome) was to validate how ProCoS and its requirements perform under Windows 10. To do so, we conducted various resources regarding windows operating system security and finally (re-)used an existing set of windows security test provided by devsec.io[5].

All in all, we created a baseline consisting out of 9 tests, each containing at least two sub-controls (more fine-grained test). To import those *baseline tests* one needs to use the CMT-CLI which was presented in Section 7.3.

[5]https://github.com/dev-sec/windows-baseline

Discussion

The application of this phase, i.e., *CCR-Analysis* and *CCR-Definition*, revealed that the provided installation documents build a great fundamental source to derive first compliance requirements for ProCoS. However, due to their different designated usage, i.e., describing an installation process nor specific infrastructure settings, these documents yield some pitfalls when analyzing those under certain other viewpoints. More precisely, we struggled with the lack of information, i.e., linguistic remediation [Rup+14]. For example, various documents described port settings, e.g., on requires that TCP port `138` is opened for an NetBIOS service. Treating this requirement from a *compliance* perspective of view, the installation, configuration and execution of NetBIOS is demanded, consequently at least *four* instead of *one* demand is formulated. Being aware of the unconscious linguistic remediation of information, it is possible to derive compliance requirements using these installation documents very well.

Interim Conclusion

For an interim conclusion of the application of this phase, one can say, that we were able to successfully derive and create a set of compliance requirements for ProCoS. However, it is to emphasize that we only evaluated a small subset of this phase due to personal absence. Nevertheless, we were able to prepare the required outcome of this phase to proceed with the next phase of our described *Continuous Compliance* process.

Using the presented tooling (CMT) we were able to (digitally) create and define the previously outlined compliance requirements, whereas the designated tool support for this phase can be approved.

8.2.2. Continuous Compliance – Develop

In this phase, we attempt to model the *Product* and to create the necessary *Tests* to ensure the previous defined *Compliance*. Depending on the chosen proceeding, e.g., using many or even no *Placeholder*, for *Test* creation different modeling attempts occur.

In the following, we distinguish between three different attempts we have taken to model the previously defined compliance requirements in the context of ProCoS. The reason for each attempt, their application and consequence are explained in detail.

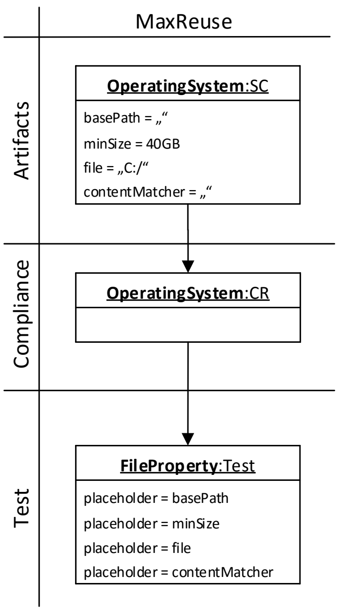
To express the difference between those attempts, we explain the realization of each by one compliance requirement, i.e, *ProCoS_Install #2*. This requirement can be formulated more intuitively as follows: "*The system main partition <u>is named C:</u> and has at least a <u>size of 40GB.</u>*". As indicated this compliance requirement enfolds two different and independent demands.

Modeling ProCoS Requirements – Maximum Reuse

The first modeling approach which resulted during our tool evaluation is named *Maximum Reuse*. Its pursued attempt was to yield as much reusability of *Tests* as possible, which resulted due to the adaptation of this phase with a traditional *developer view*. For example, we tried to reduce the needed tests to a minimum using *Test Genericity*.

Application To create as less test as possible, we classified all complied compliance requirements in resource categories, that is, we concentrated on the most fundamental properties to be checked/verified for each *Compliance*. Accordingly, for the exemplary requirement, we created a *FileProperties* test which is able of verifying any potential property of a file, e.g., its content, size or permission mode, which is shown in Figure 8.1.

Discussion Pursuing this attempt we were able to reduce all 35 compliance requirements to a tiny set of only 7 *Tests*. However, the high usage of *Test Genericity* leads to huge amount of *Placeholders* per *Test*, on average 4 *Placeholder* per *Test*. To finally generate and execute those test each of these *Placeholder* needs to be defined via an *PlaceholderValue*. Unfortunately, due to the high test abstraction, a exact context is required to accomplish this instantiation. As a consequence, a great knowledge on the actual test functionality and the exact effect of each *Placeholder* was needed in the *Artifcat* domain. This is illustrated in the object diagram, i.e., the instantiated model, in Figure 8.1.



```
describe file('<% basePath %><% file %>') do

  it{ should exist }

  if !'<% minSize %>'.empty?
    its('size'){ should be > <% minSize %> }
  end

  if !'<% contentMatcher %>'.empty?
    its('content'){
      should match <% contentMatcher %>
    }
  end

end
```

Figure 8.1.: Partial Model instance of *Maximum Reuse* and its implementation of the *FileProperty* Test

As one can track, we neglected the fundamental principle of test atomicity (uniqueness) and used only one *Test* to ensure two different compliance requirements. Thus, the overall traceability gets even harder, since it is not directly clear which demanded *Compliance* are not satisfied (*size* or *naming*) if this test fails in the certain context.

To conclude one can say that this approach of modeling and implementation is not optimal as crucial information like traceability gets lost and high knowledge regarding the exact test implementation, i.e., to correctly instantiate the *Placeholder*, is required. This first, intermediate result, led us to a more *naive* approach.

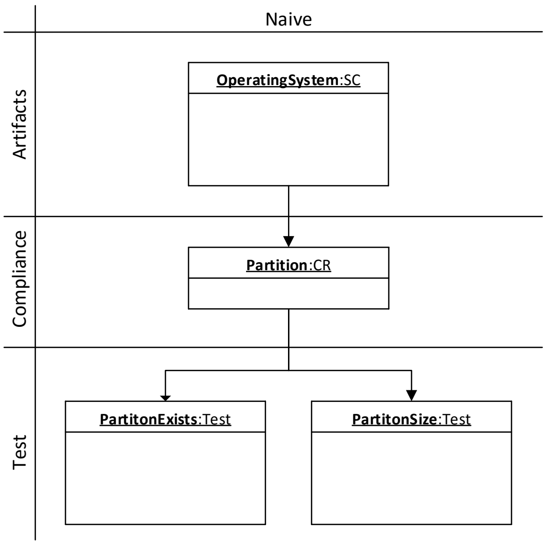
Modeling ProCoS Requirements – Naive

The basic idea of this approach was to keep the test implementation as simple as possible. To reduce the required knowledge for later modeling steps, we minded the principle of test atomicity and traceability.

Application To consider all noted properties explicitly, we ended up in creating a non-generic test for each unique compliance requirement. Thus, each test was perfectly tailored to ensure exactly one property of its origin compliance requirement.

Given the previously mentioned exemplary requirement we created two independent tests to check each of both requirements, i.e., the specific file name and size (cf. Figure 8.2).

Discussion Accomplishing this attempt for ProCoS resulted in a large set of atomic tests, i.e., we needed 38 test to ensure 35 compliance requires which is 5.5 times more *Test* entities than in the previous attempt. Furthermore, due to neglecting *Test Genericity*, we were not able to reuse once implemented *Tests*.



```
describe file('C:/') do
  it{ should exist }
end
```

```
describe file('C:/') do
  its('size'){
    should be >= 40000000
  }
end
```

Figure 8.2.: Partial Model instance of Naive approach and its implementation of the *PartitionExists* (top) and *PartitionSize* (bottom) Test

However, due to its accuracy, the attempt requires less or no expert knowledge on higher modeling domains like *Artifact* which is indicated though no instantiation of *Placeholder* (cf. Figure 8.2). Further one can exactly track the cause, i.e., the specific property, if a compliance requirement failed as every property is checked through an own *Test*.

Summarizing it is to say, that the approach is applicable and yields sufficient information to continue with the *Evaluate* phase of the overall process successfully. However, it is not optimal nor does it make use of the established modeling capabilities. As outlined above, we do not use any of the earlier crafted concepts on *Reusability* and this is why we carried out one further modeling attempt to examine if we could improve this proceeding.

Modeling ProCoS Requirements – Balanced

The naming indicates this approach should enfold a *balanced* way between the previous conducted extremes. On the one hand, this approach should keep the implementation as simple as possible and adhere to test atomicity. On the other the concepts of *Reusability* should be applied more intensively to yield a high *Reusability* and decrease code redundancy.

Application To yield atomic and reusable test we combine both previous approaches, i.e., we used the atomic tests from the *Naive* approach and extended those with *Placeholder* used in the *Maximum Reuse* attempt. For the exemplary compliance requirement, this resulted in two distinct tests (cf. Figure 8.3) since we needed to ensure two independent properties.

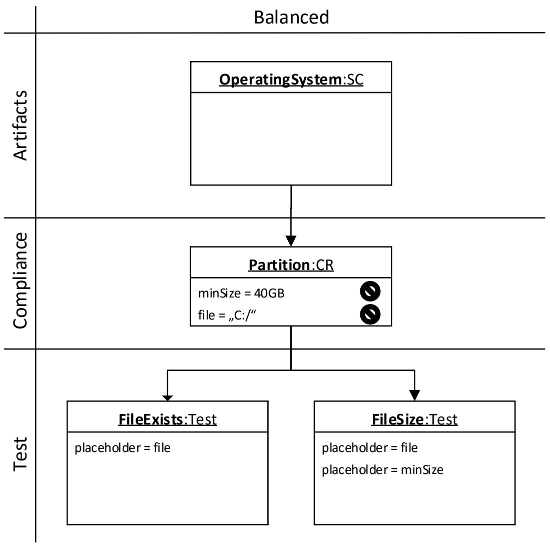
Next we defined the *Placeholder* on *Compliance* level and applied the *Blocking* concept (indicated by the black circles/dots in Figure 8.3) to prevent further redefinition.

Discussion Pursuing this modeling attempt we were able to create a high reusability of *Tests*, i.e., we only need ten tests to cover all 35 complied requirements. This further implies less coding effort.

However, we neglected the test atomicity on *Test* level, i.e., in the test code, but were able while using the *Blocking* concept to reify each test into an atomic compliance requirement on *Compliance* level. For example, the earlier used *FileProperty* test can be turned into an atomic compliance requirement ensuring that a variable file (specified by the context) with setting the placeholder *minSize* and *content* to empty and *block* further altering of those.

Nevertheless, due to the usage of *Blocking* more modeling effort is required to describe the full set of ProCoS *Compliance* demands. Further, it was somehow hard to distinguish between test- and SUT-placeholder, i.e., *Placeholder* which could be blocked or not.

Conclusively it is to say, that this modeling attempt yields the best qualities. On the one hand reusability of once implemented tests are given and on the other, the traceability is persevered since a test can be semantically reduced to check exactly one property (cf. *Blocking* in Section 6.4.1).



```
describe file('<% file %>') do
  it{ should exist }
end
```

```
describe file('<% file %>') do
  its('size') {
    should >= <% minSize %>
  }
end
```

Figure 8.3.: Partial Model instance of Balanced approach and its implementation of the *FileExists* (top) and *FileSize* (bottom) Test

Interim Conclusion

To resume all three previous presented modeling attempts, we have compiled a set of attributes describing the quality of each of those attempts in Table 8.3. The attributes *Reusability*, *Required Knowledge* and *Traceability* were already discussed beforehand. To further classify the approach we valued their *Maintenance Effort*, which is described in more detail next.

	Maximum Reuse	Naive	Balanced
Reusability	high	low	moderate
Required Knowledge	high	moderate	moderate
Traceability	-	✓	✓
Maintenance Effort	high	none	moderate

Table 8.3.: Overview of focused attributes and their characteristic for each conducted modeling attempt.

Maintenance Effort indicates the amount of review activity needed if *Compliance* and thus the appropriate *Test* implementation changes. The affected domain of each modeling attempt, which has to be reviewed due to changes, is highlighted in Figure 8.4.

For the first attempt, i.e., *Maximum Reuse*, a *Compliance* change forces a review at

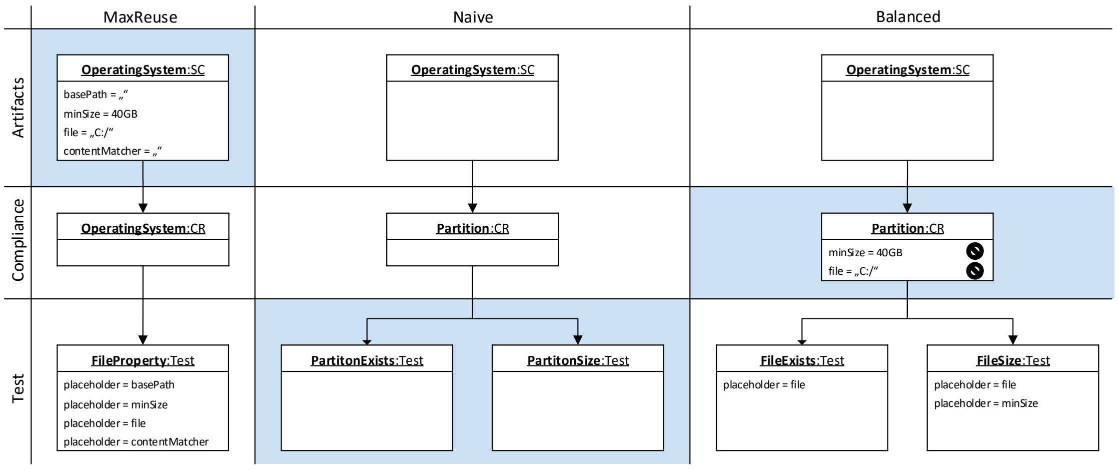


Figure 8.4.: Overview of all three modeling attempts. The light blue background highlights the affected domain when *Placeholder* change.

the highest level of abstraction, since the concrete information on that compliance is stored in the used *PlaceholderValues* on *Artifact* level. Thus, every – ideally compliance independent modeled – affected *Artifact* needs to be reviewed for correctness which results in a high *Maintenance Effort*. For example changing a very general and widely used vendor specific (but *Product* independent) *Compliance* will lead to a review of mostly every *Artifact*.

In the *Naive* approach, the *Maintenance Effort* is very low or even non-existing. This is due to the fact that each compliance requirements has its own test.

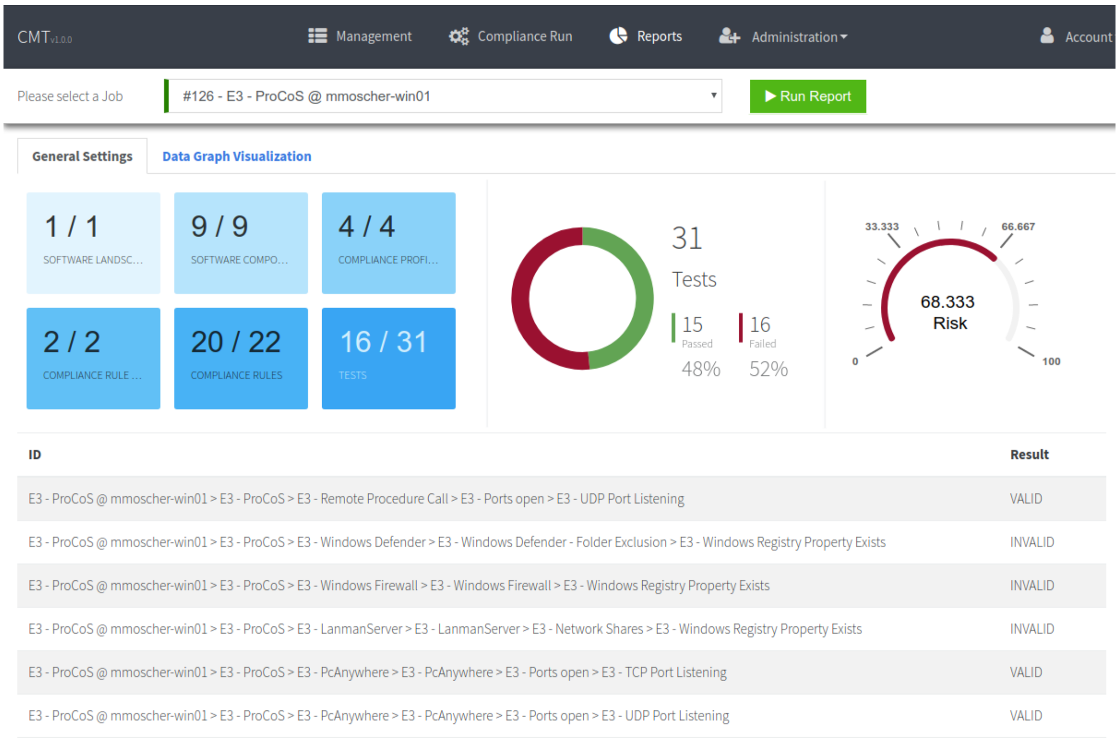
When adopting the *Balanced* approach, the *Maintenance Effort* will be somehow higher than in the *Naive* approach but even lower than in the *Maximum Reuse* attempt. Nevertheless, this amount of effort is justifiable as we tried to change some *Compliance* anyway (assumption).

To conclude, we claim that the amount of *Maintenance Effort* can be compared with the amount of used *Placeholder*. A high utilization of *Placeholder* will lead to a high *Maintenance Effort* if the a tests once changes. To this end, the utilization of *Placeholder* should be carefully considered.

Summarizing one can say that the *Develop* phase of *Continuous Compliance Testing* process can be successfully solved by different attempts. The final resulting models of each attempt can be reviewed in the appendix (cf.). These models reveal that we were successfully able to model the demanded *Compliance* and their implementing *Tests* as part of a ProCoS *Product*. To this end the *applicability* of this phase using the CMT is proven. We have exemplary presented each attempt on basis of a concrete ProCoS demand, i.e. *ProCoS_Install #2*, and discussed benefits together with certain drawbacks. However, we remain with the assumption that each attempt is valid and should yield the same results. At the bottom line, each user has to decide on its own how to model its specific compliance. As our evaluation indicated, we would suggest following the *Balanced* attempt, since it enfolds the best usage of *Reusability* and enables *Traceability* as well.

8.2.3. Continuous Compliance – Evaluate

The execution of the last phase as part of *Continuous Compliance Testing* revealed that the tool supported reporting is applicable.



ID	Result
E3 - ProCoS @ mmoscher-win01 > E3 - ProCoS > E3 - Remote Procedure Call > E3 - Ports open > E3 - UDP Port Listening	VALID
E3 - ProCoS @ mmoscher-win01 > E3 - ProCoS > E3 - Windows Defender > E3 - Windows Defender - Folder Exclusion > E3 - Windows Registry Property Exists	INVALID
E3 - ProCoS @ mmoscher-win01 > E3 - ProCoS > E3 - Windows Firewall > E3 - Windows Firewall > E3 - Windows Registry Property Exists	INVALID
E3 - ProCoS @ mmoscher-win01 > E3 - ProCoS > E3 - LanmanServer > E3 - LanmanServer > E3 - Network Shares > E3 - Windows Registry Property Exists	INVALID
E3 - ProCoS @ mmoscher-win01 > E3 - ProCoS > E3 - PcAnywhere > E3 - PcAnywhere > E3 - Ports open > E3 - TCP Port Listening	VALID
E3 - ProCoS @ mmoscher-win01 > E3 - ProCoS > E3 - PcAnywhere > E3 - PcAnywhere > E3 - Ports open > E3 - UDP Port Listening	VALID

Figure 8.5.: CMT Reporting Screen for Job-Run #126 of the balanced modeling approach (E3). Showing the *Entity Type Count*, *Compliance Index* and *Maximum Compliance Violation* (left to right). In the lower all derived *Tests* and their accurate context are listed.

Figure 8.5 illustrates the reporting and its *KPIs* for the execution of the third modeling attempt (*Balanced*, cf. Section 8.2.2). The leftmost KPI indicates an *Entity Type Count*, i.e., how many entities were modeled and how many instantiated during the test run. The *Compliance Index* is visualized in the middle. Rightmost the *Maximum Compliance Violation* is represented.

The *Compliance Index* indicates that our ProCoS testing system is not yet compliant. This is primarily due to the usage of *Windows Baselinetest* which were mentioned earlier. The presented *Compliance Violation* value does not yet have any meaning since we do not possess the appropriate expert knowledge to value each *Compliance* impact correctly.

However, further actions like *Remediation* were not conducted as part of this evaluation since this would exceed the scope of this thesis, i.e., we only focus on compliance testing and not system hardening/remediation as outlined in our problem statement (cf. Chapter 3).

8.2.4. Conclusion

To finally conclude this case study, i.e., the applicability and tool support of *Continuous Compliance Testing*, it is to say that we successfully adopted each phase and were able to create an exemplary set of *Compliance* from the *product-point-of-view* for *ProCoS*.

First in Section 8.2.1 we have executed the *Elaborate* phase and presented the derivation and definition of compliance requirements for *ProCoS* in very detail. We were able to derive 35 demands on total. However, we assume that there exist even more compliance requirements which we do not recorded due to missing expert knowledge. But for a first shot and application of the presented process model, we have demonstrated that installation documents yield a very good basis.

Next in Section 8.2.2 we applied the *Develop* phase and discussed three different approaches on how to implement and model a representative compliance requirement (*ProCoS_Install #2*). We claim that each of these approaches is valid and yield the same results. However, as discussed, varying advantages and drawbacks exist. On the basis of an exemplary requirement and the utilized the models, we have proven the practicalness of this phase.

At least, in Section 8.2.3, we shortly presented the compliance test results for the third modeling attempt. It revealed that a *standard ProCoS* installation is not (by default) compliant with our defined compliance requirements.

However, due to missing personal, we were not able to apply every task of this phase. Due to this reason, we had to rely on installation documents only and did not conduct all described roles and stakeholders. Moreover the exploration of such documents is not yet defined by our promoted process models. Such crucial modifications to the overall process adaptation have to be noted because they – primarily the omission of the defined roles – enfold a *Threat to Validity* [RH08] of our conducted case study. Nevertheless, we claim that a general tool support using the *CMT* is established.

The conclusion of this evaluation is, that the tool oriented process is applicable.

8.3. Reusability

To assess the overall produced reusability we divide this evaluation into two categories. One the one hand, the technical provided reusability, i.e., reusing entites during modeling, and on the other hand, the functional motivated reusability, i.e., instantiating *one* abstract *Test* multiple times with other values.

Technical Reusability Due to the chosen modeling attempt, i.e., the *separation of concerns* and the resulting meta model for the *Compliance Management Tooling*, a technical reusability is assured. To which extend this kind of reusability is applied, depends on the chosen modeling attempt. However, as we only evaluated one *Product* yet, we are not able conclude on the usefulness of this kind of reusability. It is to assume that large-scale evaluation will outline *inter-model* reusability.

Functional Reusability To prove the validity of the proposed (functional) *Reusability* concept, i.e., the usage of *Test Genericity*, *Whitelisting* and *Blocking* leading to *test case parametrization*, we consider the *context-sensitively* (cf. Section 7.2.3) usage of all entity types. Therefore, we compared the amount of *modeled* entities as part of an *ApplicationSystem* with the amount of *instantiated* entities, i.e., one generated test case with a specific set of test data (cf. [Mes07]). As described in the Whitelisting concept a specific *Test* can be used several times in the same contexts, i.e., it is derivable differently (cf. Section 6.4.1).

Figure 8.6 illustrates the contrast of *modeled* (left bar) and *instantiated* (right bar) entities of each conducted modeling attempt as part of the *Develop* phase evaluation. We grouped all entities in three different categories, i.e., *Test*, *Compliance Rules* and *Modeling Entities*[6], to precisely indicated the main field of reusability.

It is noticeable that the functional reusability depends on the chosen modeling attempt. As part of *Maximum Reuse* we yield a reusability of around *140%*, with the *Balanced* attempt around *30%* and with the *Naive* a reusability of *0%* (as intended). This data shows that the concept of *Reusability* is applicable and operates as assumed.

Nevertheless, it should be mentioned that we only focused the so called *intra-model* reusability since our evaluation set only comprises one product. We claim that when performing an evaluation at a larger scale, e.g., with 10 or more products modeled in the same CMT instance, an *inter-model* reusability is given. However, this is to evaluate.

To conclude this brief evaluation, it is to say, that the proposed concept of reusability is applicable (previously also indirectly assessed as part of the case study) and yields the desired results.

[6]all other entities, e.g., *Artifacts* and *Compliance Rule Sets* used in the respective model

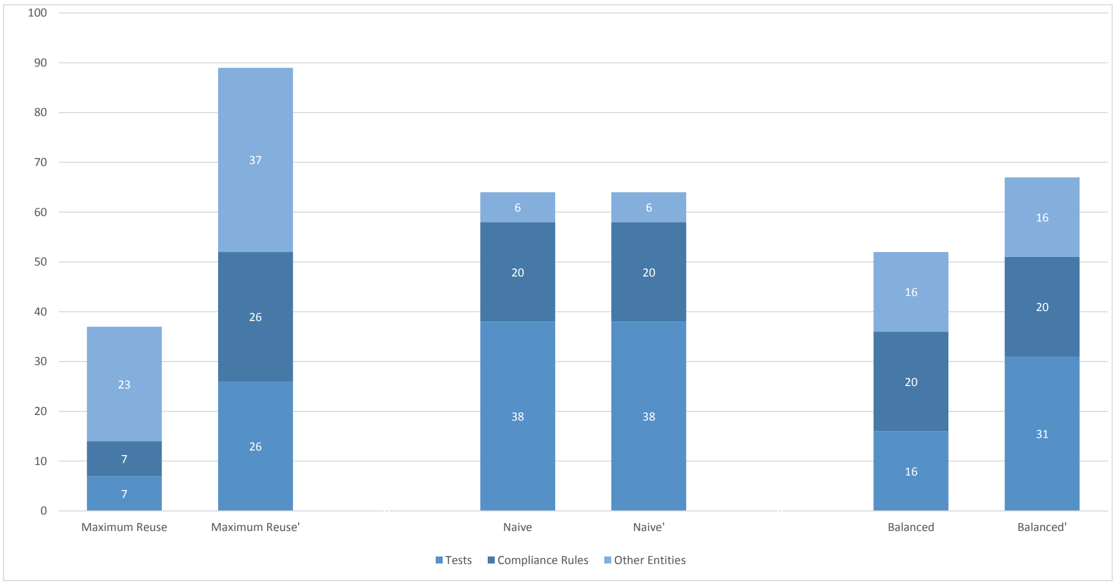


Figure 8.6.: Bar chart contrasting the *modeled* (left bar) and *instantiated* (right bar) entities of each modeling attempt.

8.4. Integration in Process Models

To evaluate the further utilization of our presented tooling, we analyzed an existing maturity model, i.e., *OpenSAMM*, a standard on information security, i.e., *BSI*, and a widely known collection of processes to ensure a proper (and secure) information infrastructure, i.e., *ITIL*. How our tooling supports each of those is briefly presented in the following.

OpenSAMM The *OpenSAMM* is a framework to help organizations formulate and implement a strategy for software security that is tailored to the specific risks facing the organization (cf. Section 2.4). To its similar domain of application, i.e., software security and risk management, we claim that our solution could assist and further simplify the adaptation of different maturity levels formulated in *OpenSAMM*. More precisely we claim to support the *Policy & Governance*, *Security Requirements*, *Security Testing* and *Environment Hardening* security practices. Next, we argue which maturity level of each security practices is supported. However, we do not recap the different objectives in detail as those can be looked up in the related document [OWAa].

Policy & Governance is focused on understanding and meeting external legal and regulatory requirements while also driving internal security standards to ensure compliance in a way that's aligned with the business purpose of the organization [OWAa].

This practice is fully supported on all three maturity levels, i.e., *PC1*, *PC2* and *PC3*. Because using *Continuous Compliance* one is able to clearly *define* (PC1) and *create* (PC2) compliance requirements. Furthermore, using the presented *CMT-CLI* an additional quality gate can be implemented (PC3).

Security Requirements is focused on proactively specifying the expected behavior of software with respect to security [OWAa].

Our process based solution is capable solving the second central *activity* of the maturity level *SR1* and *SR3*. For example necessary compliance requirements are conducted during our *Elaborate* phase (SR1B). Additional certain requirements can be mandate for products, e.g., using a quality gate in combination with the *CMT-CLI* (SR3B).

Security Testing is focused on inspection of software in the runtime environment in order to find security problems [OWAa].

The maturity levels *ST2* and *ST3* are partially solved applying the presented solution, since certain security test can and will indirectly be integrated into the development process (ST2A). Furthermore, using the *CMT-CLI* in conjunction with an automation system like Jenkins on can easily fulfill this phase.

Federal Office for Information Security (BSI) The *Federal Office for Information Security (BSI)* offers an annual updated document, i.e., the *BSI Grundschutz*, describing standards and physical provisions to yield a certifiable quality regarding IT and information security. As earlier discussed in Section 2.2.1 this standard can be used as basis for compliance testing. We present those categories (*Baustein*) of the *BSI Grundschutz* which could be partially modeled with our solution and furthermore integrated in every *Compliance*. Similar to the described integration of *OpenSAMM* (cf. Section 8.4) we do not explain the categories in greater detail, since those can be looked up online [ISa].

Baustein B3: Sicherheit der IT-Systeme describes certain recommendations on how to securely operate information system, e.g., *Windows-* or *Linux-* operating Server. However, as this catalog yields many items one has to decide itself which should or could be applied. In general one is able to check all technical described measures using our proposed solution. To this end, we claim that our tool can actively assist this *Baustein*.

Baustein B5: Sicherheit in Anwendungen similar to the previous *Bautsein* this one yields a lot of different items with a description of technical measures each. Thus, we claim that this *Baustein* can be realized alike to the previous one.

IT Infrastructure Library (ITIL) The *IT Infrastructure Library (ITIL)* represents a set of widely known and adapted *process*, *functions* and *roles* applicable to nearly any IT infrastructure to yield a well-defined and even secure standard. Due to its large extent, e.g., five main areas with 37 key processes in total, we present a possible support by our solution at a very superficial manner. We will outline the support of three major process, i.e., as part of the *ITIL-Service Design* processes. Our explanations referrer to the official ITIL processmap created by Stefan Kempter et. al [Kem].

Risk Management enfolds the main activities of identifying, assessing and controlling risk. Due to the possibility of modeling and reporting compliance violation of (failed) compliance requirements, our presented solution is suited to support the sub-process of *Risk Monitoring*. As required by this process one can monitor the ongoing process of countermeasure implementation, i.e., comparing the *Compliance Index* of the *ApplicationSystem* in a continuous manner.

Information Security Management aims to ensure the confidentiality, integrity and availability of an organizations data and IT services [Kem]. Adopting the CMT a user is able to realize the *Security Testing* and *Security Review* sub-processes. First, *Security Testing*, makes sure that all security characteristics are part of the regular testing. Since the CMT allows an external interaction, one could easily integrate compliance testing execution into the regular testing process. Further all gained results can be evaluated and used to *verify* if all measures and procedures (for security) are maintained, i.e., *Security Review*.

Compliance Management tries to ensure that certain services, processes and system comply policies and requirements. With our presented solution this process can be fully implemented, i.e., the sub-processes *Compliance-Register*, *Compliance-Review* and *Enterprise Policies and Regulations* are supported. More precisely the solution is able of creating (*register*) and executing compliance requirements. Furthermore, the gathered results can be *reviewed* at anytime.

To finalize this short excursion on further existing process models facing software security and compliance, it is to conclude that our presented process and its tooling (*Continuous Compliance Testing*) is valuable integrateable into other models. We have exemplary discussed how our solution would support certain processes or maturity levels in *OpenSAMM* and *ITIL*. Furthermore, we have shortly stated that one is able to continuously check for (partially) compliance with the BSI using our solution.

8.5. Code Quality

To briefly evaluate the code quality of our created tooling we integrated *Sonar* as part of our build pipeline. As illustrated in Figure 8.7 the overall code quality and each single measured metric, i.e., *Reliability*, *Security* and *Maintainability*, are rated with *A*. This indicates an excellent code quality and its maintainability indicates that a further ongoing development is easily achievable. Figure 8.8 shows further considered

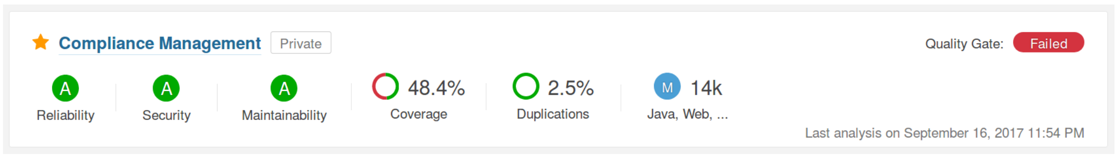
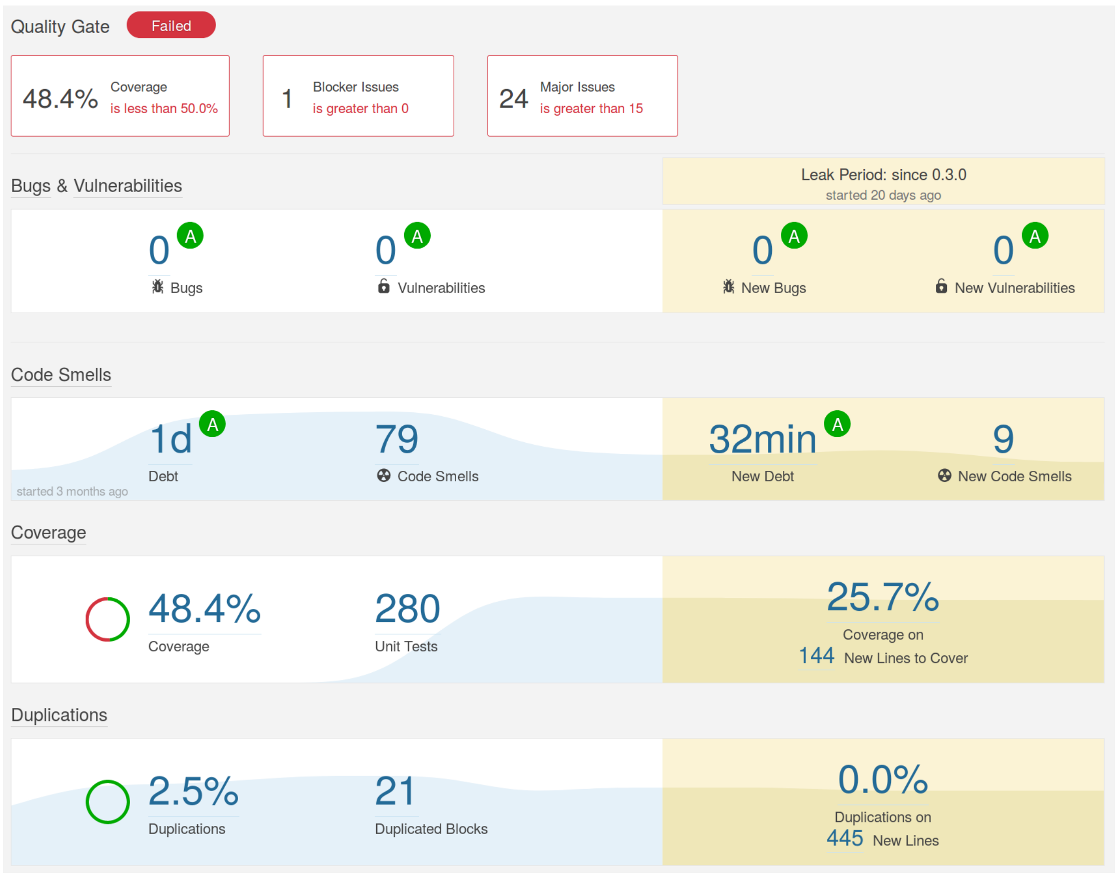


Figure 8.7.: Sonar overview screen of the *Compliance Management* project. As illustrated, the project has a excellent (A-grading) source code quality.

metrics. More precisely out code to not contains any *Blocker Issues*, *Major Issues*, *Bugs* or *Vulnerabilities*. Unfortunately, we do not yet covered enough lines of code within our prepared unit tests. However, since we only intended to provide a *proof-of-concept* implementation and not a *production-ready* software we claim that a test coverage of 48.5% by a total amount of roughly 14.000 LOC is a very adequate foundation for further development.

To conclude one can say that our submitted code for the *proof-of-concept* implementation possesses a very high code quality.



Quality Gate Failed

48.4% Coverage is less than 50.0%

1 Blocker Issues is greater than 0

24 Major Issues is greater than 15

Bugs & Vulnerabilities

Leak Period: since 0.3.0
started 20 days ago

0 A Bugs

0 A Vulnerabilities

0 A New Bugs

0 A New Vulnerabilities

Code Smells

1d A Debt

started 3 months ago

79 Code Smells

32min A New Debt

9 New Code Smells

Coverage

48.4% Coverage

280 Unit Tests

25.7% Coverage on 144 New Lines to Cover

Duplications

2.5% Duplications

21 Duplicated Blocks

0.0% Duplications on 445 New Lines

Figure 8.8.: Detailed Sonar report of the *Compliance Management* project.

8.6. Discussion

To bring the presented evaluation to a conclusion, we resume and shortly discuss the revealed results.

First and foremost in Section 8.2, we *verified* and *validated* the practical adaptation of the *Continuous Compliance* process using the presented tooling, i.e., the *Compliance Management Tooling.* The results indicated that a general applicability of the process and the tooling is possible. Furthermore, the excellent modeling capabilities revealed as part of three different realized modeling attempts. However, due to personal restriction we were not able to evaluate the roles in accurate detail and noted a possible *Threat to validity* of our case study. Nevertheless, we were able to yield first, comprehensive results for *ProCoS* which we presented as part of the last phase, i.e., *Evaluate.* To this end, we claim that the intended processes and its tool-support are applicable.

Next, we assessed the goodness of our suggested *Reusability.* We presented the intra-model reusability for each previously created modeling attempt. As it was expected the functional reusability depend on the chosen modeling approach. This leads us to the assumption that the proposed concept on *Reusability* works as intended. However, it is even possible to neglect this enhancement as we proved through the application of the *Naive* modeling attempt. It is to conclude that we successfully created a tooling which supports the postulated reusability of *Continuous Compliance.* Furthermore, we indirectly proved that reusability in the domain of *Compliance Testing* is valid and applicable, i.e., the approach of *Data Driven Testing* [Mes07] was successfully adopted to this domain.

Last in Section 8.4, we present further other procedure models and standards to enhance secure software development. We successfully showed that even other process models, i.e., not only our novel model, can be supported by the *Compliance Management Tooling.* To this end, we claim that the presented solution can be considered as excellence enhancement in the area of *Compliance Testing* and secure software development, which results due to the ability to model even product unrelated compliance tests, i.e., generally applicable one.

9. Conclusion

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

EDSGER DIJKSTRA

Contents

In this thesis, we successfully developed an iterative and incremental process model to tackle the domain of *Compliance Testing* from a *produt-point-of-view.* Based on its general domain model, we crafted a more specific meta-model to be able to precisely model each aspect, i.e., *Product*, *Compliance* and *Tests*, in great detailed. To promote a practical adaptation, we presented a blackbox framework which support each of the introduced phases. Applying the process model in conjunction with the novel blackbox framework in a case study, revealed that our solution is applicable.

Conclusively, we can approve that each of the central problems, i.e., *System Misconfiguration*, *Information Management* and *Management Inexperience* introduced in Chapter 3, are all at least partially solved. First and foremost, we assert that we have completely solved the issue of *Information Management.* Given the previously presented evaluation the invented and created *Compliance Management Tooling* is capable of creating, managing, versioning, and documenting *Compliance* requirements from a *product-point-of-view.*

The problem of *System Misconfiguration* is mainly solved since applying the *Continuous Compliance Testing* allows to model and execute *Compliance.* Thus, one is able to ensure an *ApplicationSystem* for compliance. However, as recommended by the OWASP Top 10, our approach is not yet capable of remediation or system hardening.

Last, the problem of *Management Inexperience* is only partially solved, as were are not able to entirely define and tackle management activities. However, we proposed and provide a blueprint regarding the desired process, its roles and their responsibilities. As our evaluation resulted this process model serves as a fundamental guideline to tackle the overall problem in a first approach.

9.1. Summary

In Chapter 1 we gave a brief introduction to the central topic, i.e., *Compliance Testing* from a *product-point-of-view*, and motivated our research attempt.

Next in Chapter 2, we presented the fundamental topics to comprehend this thesis, e.g., we introduced *Compliance Testing* and the *Software Development Lifecycle*.

To identify the central problems which were resolved as part of this thesis, we presented the overall problem in Chapter 3 in more detail. We classified three distinct problems, i.e., *System Misconfiguration*, *Information Management* and *Management Inexperience*.

In Chapter 4 we presented the current field of *Comppliance Testing* and *Infrastructure Compliance Automation*. We introduced various tools, discussed their enhancements and precisely argued their inadequately regarding our defined set of problems, i.e., *System Misconfiguration*, *Information Management*, and *Management Inexperience*.

Based on these foundations, we proposed our solution to the stated problem in Chapter 5. First we introduced and defined the overall process domain. We explained the general concepts of *Product*, *Compliance*, and *Test*. Furthermore, we described the reason of portioning, i.e., additional modeling layer (*Product*) and the option to reusability, and declared the *ApplicationSystem*, i.e., the resulting context for each model.Next, we presented the general process model *Continuous Compliance* to derive, create and test compliance requirements from a *product-point-of-view*. To specify each phase in more detail, we discussed and claimed roles and responsibilities which are required to derive compliance requirements for an SUT successfully.

To promote the practical application of this process we introduced a technically motivated concept in Chapter 6. We defined the concept of *Continuous Compliance Testing* and explained the mapping towards the previously introduced *Continuous Compliance* process model. Furthermore, we described software requirements for a tool supporting the process and outlined our novel ideas for the *Compliance Management Tooling* in Section 6.2. To conclude this definition we defined a meta model which enfolds all necessary aspects to create a tooling for our process model successfully, i.e., the *Software Meta-model* (cf. Section 6.3). Based on this foundation, we defined further technical motivated concepts, such as *Reusability* and *Reporting*.

In Chapter 7 we presented a *proof-of-concept* realization for the *Compliance Management Tooling*. First, we crafted a blueprint software architecture which relies on the *CMT meta model* defined in Chapter 6. Given that architecture, we describe central aspects of our implemented, e.g., we presented the adaptation of the *Visitor* and *Strategy* pattern. Furthermore, we outlined the key-points taken to realize of *Reusability* and *Reporting*.

Finally, in Chapter 8, we evaluated our contributions. We *verified* and *validated* the *Compliance Management Tooling*. The major result is that our developed tooling is capable of supporting the created *Continuous Compliance* process model. Furthermore, we successfully evaluated the usage of *Reusability* and *Reporting*.

It is to conclude, that we successfully established a tool-supported process to yield *Continuous Compliance Testing* from a *product-point-of-view*.

9.2. Future Work

To continue and further enhance our presented solution on *Continuous Compliance Testing* from a *product-point-of-view* we describe some topics. These topics mainly accrued during our evaluation and discussion with involved experts at KISTERS.

Portability First and foremost an enlarged *Portability* of the generated tests and an executable run time would improve and encourage usage of the *Compliance Management Tooling.*

Presently our solution only supports testing a remote environment which is accessible through *SSH*, *WinRM* or plain password authentication. This yields uncertainty in various departments as not every customer end-system should be accessed via any remote protocol. To this end we purpose further investment in such functionality, because it would reduce the overall uncertainty regarding tool-adaptation and would yield even more application areas.

A blueprint solution is to couple the *TestRunner* and its associated plug-in even less with the CMT-Framework. Furthermore, the plug-in needs to be able execute its generated test code without relaying on external tools, e.g., it has to contain its own test runner executable.

Document generation To exploit the entire model and its derivable information, we suggest to create an additional document generation for each modeled *Customer-Project.* Similar to the graph visualization it should support the traceability of once modeled information. Furthermore, it would facilitate a simplified communication regarding existing compliance requirements.

In future the CMT should be capable of generating detailed documents, e.g., PDF's, for each modeled *CustomerProject.* For example this document should describe all derived compliance tests, outline their purpose (*description*) and the implementing test. Further to fully comprehend the context the derivation (inheritance) path and placeholder determination, i.e., each specific value and its origin, should be described in great detail.

A simple possibility to implement this feature would be to create an additional visitor which gathers all relevant informations combined with an basic document export class. For instance, the latter one could relay on *Freemarker* as well.

Large scale evaluation Our conducted evaluation and its exemplary adaptation of our proposed *Continuous Compliance* process model highlighted certain amount of uncertainty regarding the model validity and the overall reachable reusability. For example, we discover various different approaches on modeling compliance for a specific product. Each of these approaches yields another quantity of reused entities. Furthermore, since we applied our solution only to one large scale software product we were not able to claim on intra-model reusability. To this end, we highly suggest to conduct further and more extensive evaluation of our novel framework and its guideline, i.e., the defined process model.

Integration of EAM or CMDB System An integration of external *Configuration Management Database* system will reduce the required management overhead. Accurate, customer related settings, could directly be sourced from an existing system and do not need to be duplicated. Furthermore, we could imaging to adapt *Enterprise Asset Management* systems as well to store, receive and manage the modeled product information, i.e., *Product* and *Compliance*.

CMT-DSL We propose the creation of an additional *Domain Specific Language* to simplify and quicken the model creation and evaluation, i.e., test execution. Such an DSL should have the ability to represent the required compliance and artifact entites, as well as their relation among each other. However, pursuing the approach required additional thoughts on intra-model relations, e.g., inheriting already defined compliance or artifact entities. Furthermore the provided *Command-Line-Interface* needs to be extended.

Summarizing it is to say, that our work builds an enormous foundation for several further topics in the field of *Continuous Compliance*. Future work can be very *practical*, e.g., implementing additional features like *Portability*, or more academic oriented, e.g., discussing and evaluating further *large scale* evaluation.

A. Model Derivation Graphs

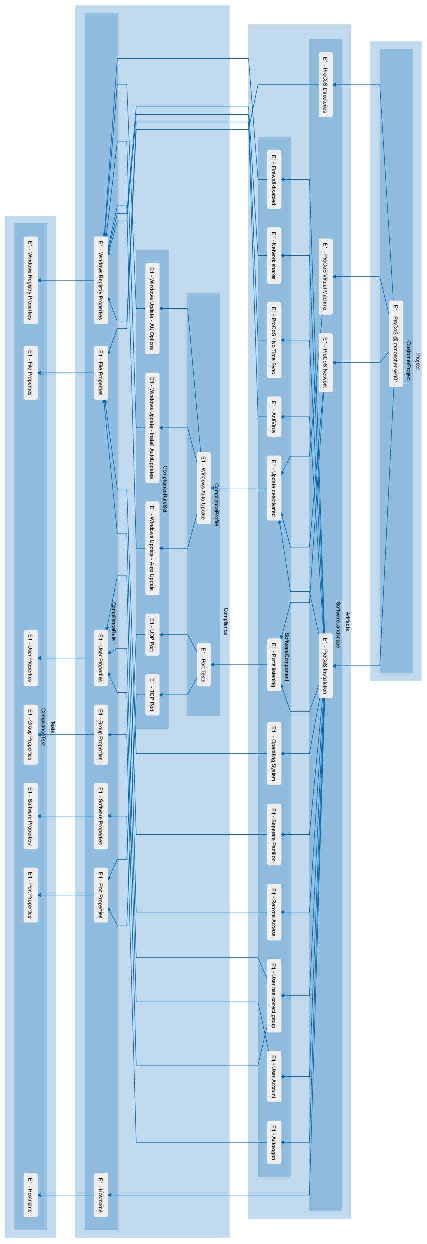


Figure A.1.: Derivation Graph (Visualization) of modeling attempt *Maximum Reuse*

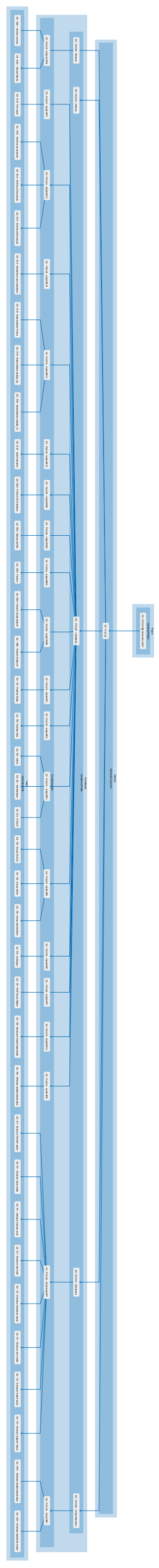


Figure A.2.: Derivation Graph (Visualization) of modeling attempt *Naive*

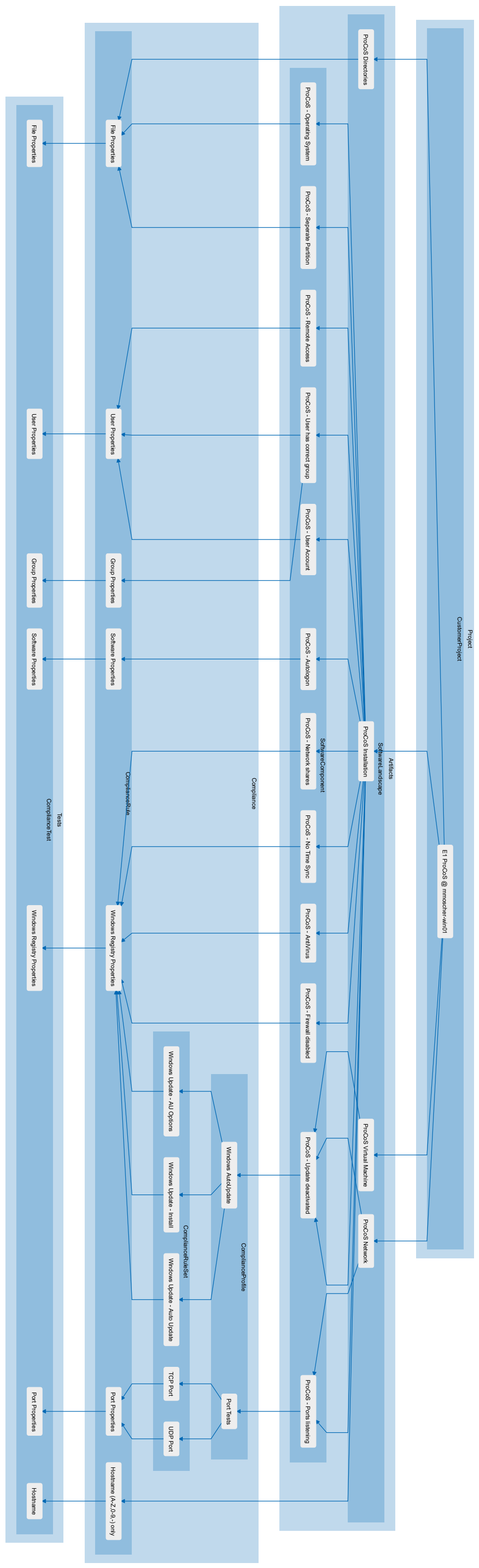


Figure A.3.: Derivation Graph (Visualization) of modeling attempt *Balanced*

Bibliography

[And10] R. J. Anderson. *Security engineering: a guide to building dependable distributed systems.* John Wiley & Sons, 2010 (cited on page 20).

[Bir] J. Bird. *Compliance as Code.* URL: `https://www.oreilly.com/learning/compliance-as-code` (visited on 03/14/2017) (cited on page 19).

[Bir16a] J. Bird. *DevOpsSec: securing software through continuous delivery.* O'Reilly Media, 2016 (cited on pages 8, 19).

[Bir16b] J. Bird. *DevOpsSec: securing software through continuous delivery. Development Operations Security.* Sebastopol, CA: O'Reilly Media, 2016. URL: `http://cds.cern.ch/record/2213288` (cited on page 20).

[Boo05] G. Booch. *The unified modeling language user guide.* Pearson Education India, 2005 (cited on page 43).

[Che16a] Chef. *Chef Automate - Deliver Software at Speed.* 2016. URL: `https://www.chef.io/automate/` (visited on 07/26/2017) (cited on page 23).

[Che16b] Chef. *InSpec - Audit and Test Framework.* 2016. URL: `http://inspec.io/` (visited on 03/17/2017) (cited on pages 9, 15, 19).

[Cir16] S. Cirulli. *Continuous Security with Jenkins and Docker Bench.* Aug. 31, 2016. URL: `https://sandrocirulli.net/continuous-security-with-jenkins-and-docker-bench/` (visited on 07/25/2017) (cited on page 23).

[Com09] O. Community. *Open Source vulnerability scanner and manager.* 2009. URL: `http://www.openvas.org/` (visited on 07/25/2017) (cited on page 23).

[Cou] P. S. S. Council. URL: `https://de.pcisecuritystandards.org/minisite/en/` (visited on 05/30/2017) (cited on page 7).

[Den05] P. J. Denning. “The Locality Principle”. In: *Commun. ACM* 48.7 (July 2005), pp. 19–24. ISSN: 0001-0782. DOI: `10.1145/1070838.1070856`. URL: `http://doi.acm.org/10.1145/1070838.1070856` (cited on page 54).

[Dij82] E. W. Dijkstra. “On the role of scientific thought”. In: *Selected writings on computing: a personal perspective.* Springer, 1982, pp. 60–66 (cited on page 64).

[Doc16] Docker. *Docker Bench for Security.* 2016. URL: `https://dockerbench.com/` (visited on 07/25/2017) (cited on page 23).

[DS16] J. Dubios and D. K. Sasidharan. *JHipster - Generate your Spring Boot + Angular apps!* 2016. URL: `https://jhipster.github.io/` (visited on 08/21/2017) (cited on pages 68, 69).

[Fow02] M. Fowler. *Patterns of enterprise application architecture.* Addison-Wesley Longman Publishing Co., Inc., 2002 (cited on page 64).

[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 65, 94).

[Gam+95] E. Gamma et al. *Design Patterns.* Vol. 47. Addison Wesley Professional Computing Series February. 1995, pp. 1–429 (cited on pages 71, 77).

[Gar15] D. Garey. *System Misconfigurations Can Put Your Data at Risk.* tenable, 2015. URL: `https://www.tenable.com/blog/system-misconfigurations-can-put-your-data-at-risk` (visited on 06/31/2017) (cited on pages 1, 15).

[GFL06] GFLewis. *RUP disciplines.* 2006. URL: `https://commons.wikimedia.org/wiki/File:RUP_disciplines_greyscale_20060121.svg` (visited on 07/01/2017) (cited on page 11).

[Har17] C. Hartmann. *Using meta-profiles with Chef Compliance.* 2017. URL: `http://lollyrock.com/articles/chef-compliance-meta-profiles/` (visited on 04/03/2017) (cited on page 22).

[Hue12] M. Huettermann. *DevOps for developers.* Apress, 2012 (cited on page 1).

[Hus16] T. Hussung. *What Is the Software Development Life Cycle?* 2016. URL: `https://online.husson.edu/software-development-cycle/` (visited on 07/01/2017) (cited on page 10).

[ISa] G. F. O. for Information Security. *IT-Grundschutz - B 1 Übergreifende Aspekte.* URL: `https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzKataloge/Inhalt/Bausteine/B1uebergeordneteAspekte/b1uebergeordneteaspekte_node.html` (visited on 09/11/2017) (cited on page 110).

[ISb] G. F. O. for Information Security. *IT-Grundschutz-Kataloge.* URL: `https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzKataloge/itgrundschutzkataloge_node.html` (visited on 03/15/2017) (cited on pages 8, 15).

[IS17] C. for Internet Security. *CIS Benchmarks.* 2017. URL: `https://benchmarks.cisecurity.org/` (visited on 03/15/2017) (cited on pages 8, 15).

[Jan10] W. Jansen. *Directions in security metrics research.* Diane Publishing, 2010 (cited on pages 21, 41, 56).

[Jel00] G. Jelen. “SSE-CMM security metrics”. In: *NIST and CSSPAB Workshop.* 2000 (cited on page 60).

[Kan02] S. H. Kan. *Metrics and Models in Software Quality Engineering.* 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201729156 (cited on page 57).

[Kem] S. Kempter. *ITIL Processes.* URL: `https://wiki.en.it-processmaps.com/index.php/ITIL_Processes` (visited on 09/11/2017) (cited on pages 110, 111).

[LL13] J. Ludewig and H. Lichter. *Software Engineering : Grundlagen, Menschen, Prozesse, Techniken.* 2013, p. 665. ISBN: 9783864900921. URL: `http://www.opac.fau.de/InfoGuideClient.uersis/start.do?Login=wouer20{\&}Query=540={\%}22978-3-86490-092-1{\%}22` (cited on pages 6, 10, 11).

[McG16] T. McGonagle. *CompOps: Continuous Delivery Needs Continuous Compliance.* 2016. URL: `https://www.cloudbees.com/blog/compops-continuous-delivery-needs-continuous-compliance` (visited on 03/09/2017) (cited on page 19).

[Mes07] G. Meszaros. *xUnit test patterns: Refactoring test code.* Pearson Education, 2007 (cited on pages 6, 54, 108, 114).

[Mic04] Microsoft. *Security Development Lifecycle.* 2004. URL: `https://www.microsoft.com/en-us/sdl/default.aspx` (visited on 08/01/2017) (cited on page 12).

[MO16] V. Mohan and L. B. Othmane. “SecDevOps: Is It a Marketing Buzzword? - Mapping Research on Security in DevOps”. In: *2016 11th International Conference on Availability, Reliability and Security (ARES)*. 2016, pp. 542–547. DOI: 10.1109/ARES.2016.92 (cited on page 20).

[NSM14] T. Netflix Security Monkey. *Announcing Security Monkey—AWS Security Configuration Monitoring and Analysis*. 2014. URL: http://techblog.netflix.com/2014/06/announcing-security-monkey-aws-security.html (visited on 07/25/2017) (cited on page 25).

[Pau93] M. Paulk. “Capability maturity model for software”. In: *Encyclopedia of Software Engineering* (1993) (cited on page 13).

[PM04] B. Potter and G. McGraw. “Software security testing”. In: *IEEE Security Privacy* 2.5 (2004), pp. 81–85. ISSN: 1540-7993. DOI: 10.1109/MSP.2004.84 (cited on page 6).

[Poh10] K. Pohl. *Requirements Engineering: Fundamentals, Principles, and Techniques*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 3642125778, 9783642125775 (cited on pages 41, 94).

[Red08] RedHat. *OpenSCAP - Audit, Fix and be Merry*. 2008. URL: https://www.open-scap.org/ (visited on 03/17/2017) (cited on pages 19, 24).

[RH08] P. Runeson and M. Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: *Empirical Software Engineering* 14.2 (2008), p. 131. ISSN: 1573-7616. DOI: 10.1007/s10664-008-9102-8. URL: https://doi.org/10.1007/s10664-008-9102-8 (cited on pages 94, 107).

[Rup+14] C. Rupp et al. *Requirements-Engineering und Management: Aus der Praxis von klassisch bis agil*. Carl Hanser Verlag GmbH Co KG, 2014 (cited on page 100).

[Sav07] R. M. Savola. “Towards a Taxonomy for Information Security Metrics”. In: *Proceedings of the 2007 ACM Workshop on Quality of Protection*. QoP ’07. Alexandria, Virginia, USA: ACM, 2007, pp. 28–30. ISBN: 978-1-59593-885-5. DOI: 10.1145/1314257.1314266. URL: http://doi.acm.org/10.1145/1314257.1314266 (cited on pages 41, 60).

[Sch15] C Schneider. “Security DevOps-staying secure in agile projects”. In: *OWASP AppSec Europe,(Amsterdam, Netherlands)* (2015) (cited on page 19).

[Spr+10] J. Sprinkle et al. “3 Metamodelling”. In: *Model-Based Engineering of Embedded Real-Time Systems: International Dagstuhl Workshop, Dagstuhl Castle, Germany, November 4-9, 2007. Revised Selected Papers.* Ed. by H. Giese et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 57–76. ISBN: 978-3-642-16277-0. DOI: `10.1007/978-3-642-16277-0_3`. URL: `https://doi.org/10.1007/978-3-642-16277-0_3` (cited on pages 43, 48).

[ST09] N. I. of Standards and Technology. *The Security Content Automation Protocol (SCAP).* 2009. URL: `https://scap.nist.gov/` (visited on 04/03/2017) (cited on pages 21, 22).

[Sto15] A. Storms. “How security can be the next force multiplier in devops”. In: *RSAConference,(San Francisco, USA)* (2015) (cited on pages 19, 20).

[Tec] TechTarget. *Conformance Testing.* URL: `http://searchsoftwarequality.techtarget.com/definition/conformance-testing` (visited on 05/30/2017) (cited on page 7).

[Ten14] Tenable. *Nessus v6 SCAP Assessments.* Tech. rep. 2014. URL: `http://static.tenable.com/documentation/Nessus_v6_SCAP_Assessments.pdf` (cited on page 24).

[Ten16] Tenable. *Nessus Vulnerability Scanner.* 2016. URL: `https://www.tenable.com/products/nessus-vulnerability-scanner` (visited on 07/26/2017) (cited on pages 19, 23).

[Ten17] Tenable. *Nessus Compliance Checks.* Tech. rep. 2017. URL: `https://support.tenable.com/support-center/nessus_compliance_checks.pdf` (visited on 07/26/2017) (cited on page 24).

[Tis] J. Tischart. *How is Security Integrated into DevOps – DevOpsSec, SecDevOps, or DevSecOps?* URL: `https://securingtomorrow.mcafee.com/business/cloud-security/security-integrated-devops-devopssec-secdevops-devsecops/` (visited on 03/14/2017) (cited on page 20).

[UpG] UpGuard. *UpGuard - Cyber Resilience Platform.* URL: `https://www.upguard.com/` (visited on 03/11/2017) (cited on page 15).

[VHS03] R. B. Vaughn, R. Henning, and A. Siraj. “Information assurance measures and metrics - state of practice and proposed taxonomy”. In: *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the.* 2003, 10 pp.–. DOI: `10.1109/HICSS.2003.1174904` (cited on page 41).

[VO10] M. Vaishnavi and L. ben Othmane. *SecDevOps: Is It a Marketing Buzzword?* Tech. rep. Technische Universität Darmstadt, 2010. URL: `https://www.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_CASED/Publikationen/2010/SecDevOps.pdf` (cited on page 19).

[Von16] S. Vonnegut. *4 Keys To Integrating Security into DevOps.* 2016. URL: `https://www.checkmarx.com/2016/07/01/201607014-keys-to-integrating-security-into-devops/` (visited on 07/12/2017) (cited on page 20).

[Wat99] J. Wateridge. “The role of configuration management in the development and management of Information Systems/Technology (IS/IT) projects”. In: *International Journal of Project Management* 17.4 (1999), pp. 237 –241. ISSN: 0263-7863 (cited on page 1).

[Win05] M. Winter. *Methodische objektorientierte Softwareentwicklung: Eine Integration klassischer und moderner Entwicklungskonzepte.* 2005, p. 540. ISBN: 3-89864-273-9 (cited on pages 10, 11).

[Wol94] P. Wolfgang. “Design patterns for object-oriented software development”. In: *Reading Mass* (1994) (cited on page 65).

[Cen16] Center for Internet Security. *CIS Benchmark v1.1.0 for CIS Docker Community Edition Benchmark.* 2016. URL: `https://www.cisecurity.org/cis-benchmarks/` (visited on 07/25/2017) (cited on page 23).

[Cen17] Center for Internet Security. *CIS Benchmarks.* 2017. URL: `https://www.cisecurity.org/cis-benchmarks/` (visited on 07/31/2017) (cited on pages 8, 21).

[Goo10] Google. *Angular.io - One framework. Mobile & desktop.* 2010. URL: `https://angular.io/` (visited on 08/21/2017) (cited on page 68).

[Gos13] Gosuke Miyashita. *Serverspe c.* 2013. URL: `http://serverspec.org/` (visited on 03/17/2017) (cited on pages 15, 25).

[IT] IT Infrastructure Library. *IT Infrastructure Library - Roles.* URL: `https://wiki.en.it-processmaps.com/index.php/ITIL_Roles` (visited on 08/06/2017) (cited on page 35).

[Int] International Software Testing Qualifications Board. *ISTQB Standard glossary of terms used in Software Testing.* URL: `http://glossar.german-testing-board.info/{\#}eng` (visited on 05/30/2017) (cited on page 7).

[Net14] Netflix. *Netflix Security Monkey, GitHub.* 2014. URL: `https://github.com/Netflix/security_monkey` (visited on 07/25/2017) (cited on page 25).

[Nor11] Normation. *Rudder - Continuous configuration for effective compliance.* 2011. URL: `https://github.com/Normation/rudder/` (visited on 07/25/2017) (cited on page 24).

[Nor16] Normation. *Rudder - Continuous Configuration and Auditing.* 2016. URL: `https://www.normation.com/en/` (visited on 07/25/2017) (cited on page 24).

[OWAa] OWASP Foundation. "OpenSAMM Core Model". In: 1.5 (). URL: `https://www.owasp.org/images/6/6f/SAMM_Core_V1-5_FINAL.pdf` (visited on 05/12/2017) (cited on pages 13, 14, 109, 110).

[OWAb] OWASP Foundation. *OWASP - Top 10 2013-A5-Security Misconfiguration.* URL: `https://www.owasp.org/index.php/Top_10_2013-A5-Security_Misconfiguration` (visited on 03/09/2017) (cited on pages i, 1, 15).

[OWA17a] OWASP Foundation. *OpenSAMM.* 2017. URL: `http://www.opensamm.org` (visited on 05/12/2017) (cited on pages 13, 34).

[OWA17b] OWASP Foundation. *OWASP Secure Software Development Lifecycle Project(S-SDLC).* 2017. URL: `https://www.owasp.org/index.php/OWASP_Secure_Software_Development_Lifecycle_Project` (visited on 08/01/2017) (cited on page 12).

[Piv02] Pivotal. *Spring-Framework Documentation.* 2002. URL: `https://spring.io/docs/reference` (visited on 08/21/2017) (cited on page 68).

[Sof] Software Testing Help. *What is Compliance Testing (Conformance testing)?* URL: `http://www.softwaretestinghelp.com/what-is-conformance-testing/` (visited on 05/30/2017) (cited on page 7).

[Tec17] TechTarget. *TechTagret Definitions by Alphabet.* 2017. URL: `http://whatis.techtarget.com/definition` (visited on 08/06/2017) (cited on page 34).

Glossary

BM Business Manager

CaC Compliance as Code

CISO Chief Information Security Officer

CLI Command Line Interface

CMM Capability Maturity Model

CMMI Capability Maturity Model Integration

CMS Compliance Management System

CRUD Create, Read, Update and Delete

CSO Chief Security Officer

IaC Infrastructure as Code

J2EE Java Platform, Enterprise Edition

JDL JHipster Domain Language

KPI Key Performance Indicator

OVAL Open Vulnerability and Assessment Language

OWASP Open Web Application Security Project

PCI-DSS Payment Card Industry Data Security Standard

PO Product Owner

QA Quality Assurance

REST Representational State Transfer

RUP Rational Unified Process

S-SDLC	Secure Software Development Life Cycle
SCAP	Security Content Automation Protocol
SD	Software Developer
SDL	Security Development Lifecycle
SDLC	Software Development Life Cycle
SPA	Single-Page Application
SUT	System Under Test
UUT	Unit under test
XCCDF	eXtensible Configuration Checklist Description Format
XP	Extreme Programming