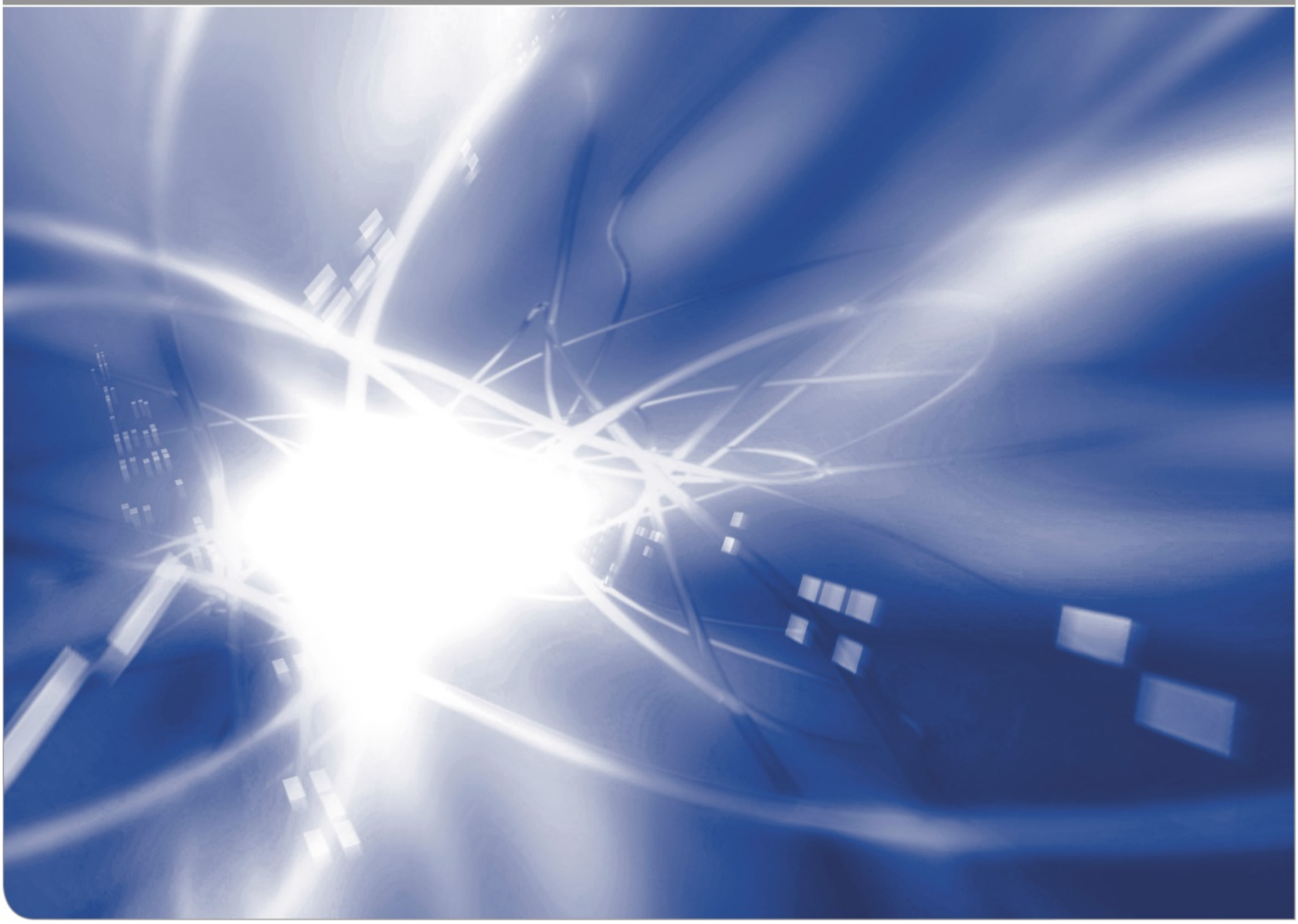
Karlsruhe Institute of Technology

Supplementary Material for the Evaluation of the Publication – A Layered Reference Architecture for Model-based Quality Analysis

by Sandro Koch[1], Robert Heinrich[1], Ralf Reussner[1]

KIT SCIENTIFIC WORKING PAPERS 188



facultative

(1) Information on affiliation as follows:

[1] KASTEL - Institute of Information Security and Dependability

(2) Information on parallel publications in anthologies, conferences

(3) Details of projects, clients, sponsors, URLs, numbers and other project-related information is not entitled

This work was supported by the Federal Ministry of Education and Research (BMBF) under the funding number 01IS18067D, and the KASTEL institutional funding.

facultative

Institute of Information Security and Dependability (KASTEL), Prof. Ralf Reussner
Am Fasanengarten 5
76131 Karlsruhe
Germany

https://dsis.kastel.kit.edu/

Impressum

Karlsruher Institut für Technologie (KIT)
www.kit.edu

2022

ISSN: 2194-1629

Supplementary Material for the Evaluation of the Publication – A Layered Reference Architecture for Model-based Quality Analysis

Technical Report

Sandro Koch, Robert Heinrich, Ralf Reussner
{sandro.koch|heinrich|reussner}@kit.edu

01.11.2022

Contents

1. Introduction

In this technical report, we present the supplementary information for the evaluation of the *Layered Reference Architecture for Model-based Quality Analysis*. In chapter 2, we present the case studies, first in their monolithic and then in their modular form. We present installation instructions for the tools required to reproduce the evaluation results in chapter 4. In chapter 5, we provide detailed information about the evolution scenarios. The tooling and the results of the scenarios can be found online [9].

2. Case Studies

In this section, we present the four case studies we modularized according to our reference architecture for model-based analyses. The model-based analyses we used are SimuLizar, Camunda, KAMP4aPS, and SmartGrid. We built modular versions of the scenarios we extracted from the case study model-based analyses for the evaluation and we only refactored them according to the reference architecture's guidelines. We did not fix bad smells that the reference design does not address because doing so would jeopardize the evaluation's internal validity.

2.1. Software Architecture Quality Prediction – Palladio Simulator SimuLizar

The Palladio Simulator is an established software architecture quality analysis tool based on the *Palladio Component Model* (PCM). Contrary to its name, the Palladio Simulator consists of three performance analyses capable of determining the performance of software architecture: SimuCom, EventSim, and SimuLizar. Each of these analyses has a distinct set of features with different priorities. SimuCom covers most features of the PCM, it generates the analysis code based on the model, but it has performance issues for large software architectures. EventSim interprets instances of the PCM, and only supports the performance analyses of software architecture while ignoring many features of the PCM. In contrast to SimuCom, EventSim has fewer issues with large software architectures due to its event-based nature [12]. SimuLizar interprets the PCM, and it supports most of the PCM features. Due to their different approaches, their source code is not interchangeable; thus, the three analyses are incompatible. We focus on SimuLizar, as it is actively maintained. One of the main issues the developers had before the maintenance and development stopped were that changes in the PCM required changes in all three analyses. All three are historically grown model-based analyses, with the typical deterioration of the internal quality over time. SimuLizar is a historically grown model-based analysis, with the typical deterioration of the internal quality over time. Other historically grown model-based analyses show similar problems. As the quality of the analysis deteriorated, more and more effort was required to sustain all three.

2.1.1. SimuLizar Overview

The Palladio Simulator consists of three analyses (SimuLizar, SimuCom, and EventSim), each of which employs a distinct analysis approach and can make performance predictions based on the PCM. SimuLizar is the most sophisticated of the three analyses; thus, we have selected it for our case study. SimuLizar is developed since 2013; it is written in the programming language Java. SimuLizar consists of 75 packages, 306 classes, 69 interfaces, and three enums; it is divided into 36 java-projects. SimuLizar has doubled in size since 2015, with classes increasing from around 150 to over 300. It also has a long history of evolutionary changes. SimuLizar features ten openly available extensions [1] and many extensions that are not fully disclosed (e. g., student theses, experimental extensions). SimuLizar represents a historically grown and versatile model-based analysis that can analyse multiple aspects of software quality. If not stated otherwise, when we mention the term *component* we refer to *analysis component*, and when we mention the term *feature* we refer to *analysis feature*. Before the refactoring of SimuLizar, all dependencies on the metamodel PCM were consolidated in one analysis component, see fig. 2.1. We exclude the components that have no representation in the PCM due to the size of SimuLizar.

2.1.2. SimuLizar Refactoring

We started the modularisation with the release version 4.3 of the Palladio-Simulator, and used the modularised PCM presented in [6, 15]. Before we modularised SimuLizar, we had to change the dependencies of SimuLizar on the modular PCM. Changing the dependencies is necessary, as the modular PCM is not used in the Palladio-Simulator. After changing the dependencies, we analysed SimuLizar regarding the bad smells of *Language Blob* and *Feature Scatter*. We used the Language Blob bad smell to identify which classes we have to separate the components into the three desired layers. The Feature Scatter smell indicates which classes and components could be merged, as the refactoring of the Language Blobs results in many small classes. The Language Blob analysis resulted in 18 occurrences, and the Feature Scatter analysis resulted in 33 occurrences. First, we focused on the language blobs of components that are supposed to be on different layers. Therefore, we applied a horizontal-split refactoring to separate the analysis component in the layers π, Δ, and Ω, which resulted in three components. Then, we applied vertical-split refactorings to the three layers to separate the language blobs still present on these layers. The final step was to merge the components where the language features were scattered over different classes and components. We could not fix all occurrences of the Feature Scatter bad smell; for certain analysis operations, multiple language features are required. The model observing part of SimuLizar requires the *modelobserver* language feature and the *software usage* language feature. This resulted in nine components on π, 22 components on Δ, and one component on Ω. The component count increased from one component to 32 components. We reduced the number of Language Blobs from 18 to zero, and the number of Feature Scatters from 33 to ten. In the following sections 2.1.3.1 and 2.1.3.2, we present detailed information about the modular structure of SimuLizar after the refactoring.

2.1.3. Modular SimuLizar (mSimuLizar)

Figure 2.2 depicts the structure of SimuLizar after the modularisation. In the figure, we exclude the analysis components that have no representation in the language, e.g. events, the interpreter component, or the reconfiguration component, as most analysis

[1] https://sdqweb.ipd.kit.edu/wiki/SimuLizar

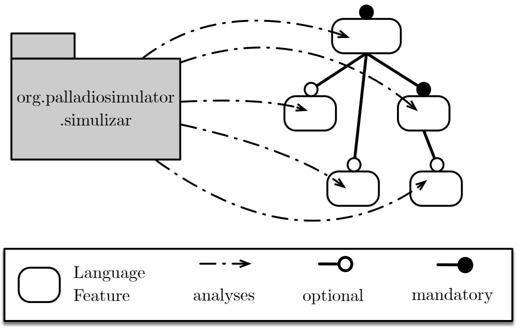


Figure 2.1.: SimuLizar dependencies on mPCM, simplified

components have dependencies on them. Including these additional components renders the already complex figure unintelligible.

2.1.3.1. Paradigm

Composition: The *composition component* handles the assembly of resources of the PCM. On the paradigm layer, the functionality of the composition component is prepared to handle any type of resources. The assembly of component types includes the preparation of resources. Preparing a resource means, setting the context and the context hierarchy of the resource. The composition component provides functionality for adding or deleting a resource and it also provides the connectors required to compose resources.

Constants: The *constants component* provides the constants required throughout the analysis of PCM instances.

Repository: The *repository component* on the paradigm layer manages the roles defined in the PCM. The PCM defines required and provided roles for components. In this component, the roles, e.g. provided and required roles, are managed. It provides interfaces to receive these roles, and also it provides interfaces to receive the signatures defined in the PCM. The main portion of the repository component is the *repository switch*. The switch contains the interpretation of the roles. It also contains the analysis code concerning the required and provided roles. The signatures are implicitly used throughout the analysis code.

Runtimestate: The *runtimestate component* provides abstract classes and interfaces for managing the state of the analysis. It holds the PCM instance, the event notification helper, and a registry of the analysed components. The *component registry* is an interface for validating whether a component is available for the analysis. It also provides add and fetch operations for the PCM components. The *event notification helper* is an interface for firing events and removing listeners.

Seff: The *Service Effect Specification* (SEFF) in the PCM represents the basic actions of a component. The *seff component* provides the interpretation and the analysis code for the elements of the seff language feature of the PCM. The seff component contains the interpreter for the seff types. For each seff type, the seff component contains the analysis code required for the elements.

Usage: The *usage component* provides the handling of probabilities defined in the usage language feature of the PCM. Probabilities are required when the analysis encounters a

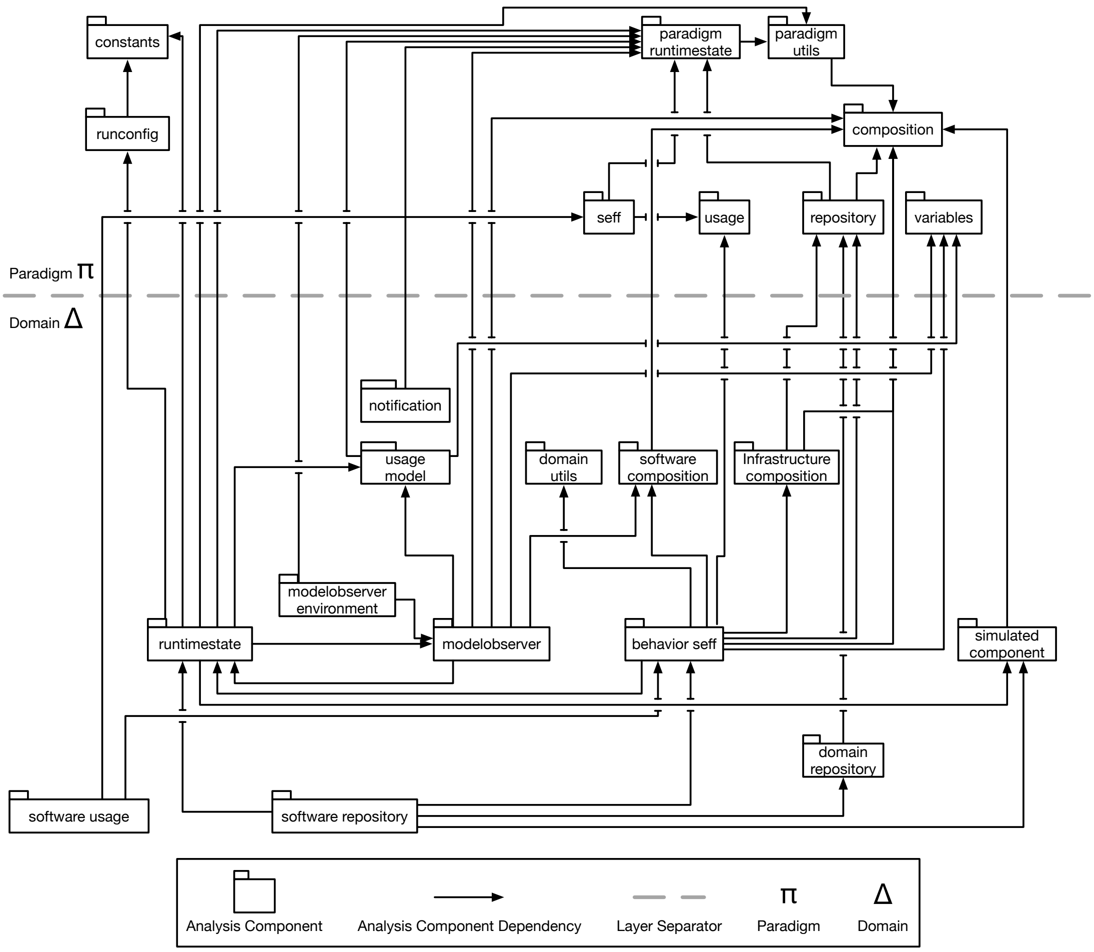


Figure 2.2.: mSimuLizar structure refactored, simplified

branch. The usage component determines in which direction the analysis must proceed. Besides branches, the usage component also provides the scheduling of delays. Another part of the usage component is the handling of loops. Based on the size of a loop, it determines the time required to finish the loop. Furthermore, the usage component provides an interface to manage user actions.
Variables: The *variables component* provides the evaluation of the model instance. It creates an evaluator instance containing the variable characterisation of the PCM and the model evaluator. The evaluation provides a condition checker, which checks whether a boolean expression in a condition holds. The variable component also provides the generation of random variables.

2.1.3.2. Domain

Behaviour seff: The *behaviour seff* component provides the analysis code for the PCM model elements *external call action*, *acquire action*, *collection iterator action*, *set variable action*, and *release action*. The analysis code requires information about the *infrastructure*; thus, in this component, remain dependencies on the infrastructure language feature. The behaviour seff component also provides analysis code which determines probabilistic transitions when encountering branches.

Domain repository: The *domain repository* component provides an interface for implementing the analysis code for the PCM model elements *provided role* and *signature*.

Infrastructure composition: The *infrastructure composition* component provides the analysis code for the PCM model elements *assembly infrastructure connector* and *required infrastructure delegation connector*. The component utilises the composition and repository component of the π layer.

Modelobserver: The *modelobserver* component provides the analysis code for the PCM model elements *communication link resource specification*, *linking resource*, *processing resource specification*, *resource container*, *workload*, *closed workload*, *open workload*, and *usage scenario*. The component requires, in addition to the modelobserver language feature, the *software usage* language feature, thus it holds dependencies on PCM types of these two language features.

Modelobserver environment: The *modelobserver environment* component provides the analysis code for the PCM model element *resource environment*. This component handles the modelobserver component, and it provides observers for the said model and the resource environment.

Notification: The *notification* component provides the analysis code for the PCM model elements *operation provided role*, *operation signature*, *external call action*, *entry level system call*, and *usage scenario*. This component has dependencies on four language features to perform the analysis.

Runtimestate: The *runtimestate* component provides the analysis code for the PCM model elements *resource environment*, and *assembly context*. The runtimestate component has only two dependencies on two language features, but it consolidates the state of the analysed system. It utilises direct knowledge (i.e., usage model component), or it utilises the modelobserver component to manage the runtime state of the analysis.

Simulated component: The *simulated component* provides the analysis code for the PCM model element *passive resource*. It represents two types of components mSimuLizar can analyse. The first component is a basic component that can be monitored, and it can acquire and release resources. The second component is a composite component, consisting of a set of basic components.

Software composition: The *software composition* component provides the analysis code for the PCM model elements *assembly connector*, *required delegation connector*, and *composite component*.

Software repository: The *software repository* component provides the analysis code for the PCM model elements *basic component* and *service effect specification*.

Software usage: The *software usage* component provides the analysis code for the PCM model elements *entry level system call*, *usage scenario*, and *usage switch*.

Usage model: The *simulated component* provides the analysis code for the PCM model elements *usage model*, *usage scenario*, *workload*, *closed workload*, *open workload*, *software usage package*.

2.2. Business Process Simulation – Camunda

The analysis Camunda is a workflow and simulation engine based on the *Business Process Modelling Notation 2* (BPMN2) *Domain-Specific Modelling Language* (DSML). The BPMN2 is developed by the *Object Management Group* (OMG). It is also an *International Organization for Standardization* (ISO) standard for modelling business processes. We selected Camunda as a case study because it covers the additional domain of business process analysis, and it can be used for further refactorings since, besides the standard BPMN2, it also supports the Case Management Model and Notation (CMMN 1.1) and the Decision Model Notation (DMN 1.1). Camunda is a fork of the free workflow management system Activiti, developed in 2010. In 2013 Camunda BPM was forked from Activiti as an open-source project by the company Camunda in Berlin. Our refactorings focus on the Camunda BPM Platform, which consolidates the dependencies on the metamodel. Due to the size of the Camunda BPM Platform[2] (over 500,000 lines of code), we were unable to refactor it in a reasonable time frame; therefore, we focused our refactorings on the affected analysis components and files of our scenarios.

2.2.1. Camunda Overview

The Camunda BMN Platform consists of 15 modules that also contain modules. It has 52 modules in total. The *model-api* module consolidates the dependencies on the BPMN2 metamodel. Figure 2.3 depicts the internal dependency structure of the Camunda BPM Platform. Turquoise nodes represent dependencies on *org.camunda.bpm* modules. Purple nodes represent dependencies on *org.camunda.bpm.model* modules. Black nodes represent dependencies on the remaining *org.camunda* modules.

2.2.2. Camunda Refactoring

Before we could refactor the Camunda BPM Platform, we had to adapt the dependencies of the analysis code to the modular BPMN2 DSML [6, 15]. The turquoise nodes in fig. 2.3 are the modules that had to be modified. The dependencies of the Camunda BPM Platform regarding the mBPMN2 metamodel are similar to the structure shown in fig. 2.1. In the *org.camunda.bpm.model* module are the dependencies on the mBPMN2 metamodel consolidated. As we did not refactor the whole analysis, details regarding the refactoring will be presented in chapter 5.

2 https://github.com/camunda/camunda-bpm-platform

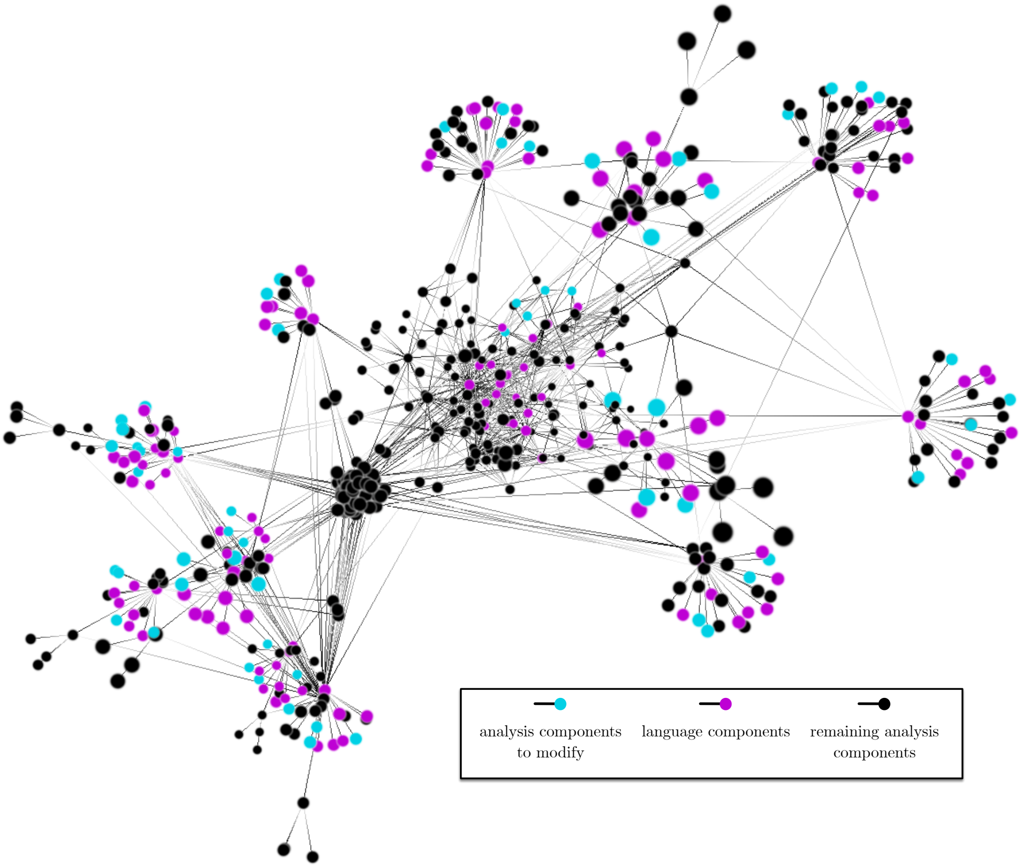


Figure 2.3.: Camunda BPM Platform Dependency Structure

2.3. Change Propagation Analysis – KAMP and KAMP4aPS

The *Karlsruhe Architecture Maintainability Prediction for Automated Production Systems* (KAMP4APS) analysis is a single-purpose analysis; its metamodel is only used by a single analysis. Although the *Karlsruhe Architecture Maintainability Prediction* (KAMP)-Frameworks methodology [5] is built to support the domains of software systems [7], business processes [14], production systems [8], and *Programmable Logic Controller* (PLC) software [2], each domain requires a dedicated analysis and metamodel. Due to the selection of the KAMP4APS metamodel in our previous work, we will focus solely on the KAMP4APS analysis. KAMP4APS covers an additional domain, which extends the diversity of our case studies. The KAMP4APS metamodel and analysis has been under development since 2016; it contains six analysis components, one of which consolidates the dependencies on the metamodel.

2.3.1. KAMP4APS Overview

KAMP4APS has a generic part that provides the framework for the change impact analysis. This framework is the foundation of the *KAMP Methodology* [5]. It provides a domain-independent part, consisting of a domain-independent modification metamodel, a set of algorithms to derive a task list, eliminate duplicates and sort elements in the task list. It also provides a metamodel and algorithms to support decision-making regarding changes in the analysed domain. Based on the domain-independent part, each domain has to provide a metamodel of the domain that will be analysed. The change impact analysis requires a structural metamodel and a non-structural metamodel. In the context of KAMP4APS the structural parts of the metamodel are the electrical and mechanical parts in a production system. Non-structural parts are, for example, documentation, drawings or tests. These models are used in the rule engine of KAMP to determine the impact of changes to the structural and non-structural parts of the system. Although KAMP4APS is separated into a domain-independent and a domain-dependent part, and the further separation into models, algorithms, structural and non-structural elements, it consists of a module containing the domain-independent part (KAMP) and a model containing the domain-specific part (KAMP4APS).

2.3.2. KAMP4APS Refactoring

Before we could refactor KAMP4APS, we had to adapt the dependencies of the analysis code to the modular KAMP4APS metamodel [6, 15]. The dependencies of the KAMP4APS regarding the modular KAMP4APS metamodel are like the structure shown in fig. 2.1. The dependencies on the modular KAMP4APS metamodel are consolidated in the KAMP4APS module. As we did not refactor the whole analysis, details regarding the refactoring will be presented in chapter 5.

2.4. Energy Network Simulation – SmartGrid

As the KAMP4aPS analysis, the SmartGrid analysis is also a single-purpose analysis. The SmartGrid energy network simulation performs an impact and resilience analysis. The metamodel is used to model topologies of smart grid energy networks. It also adds the domain of energy network analysis to our case studies; it is the second-youngest analysis, the development started in 2014. Compared to the analysis SmartGrid, the size of the SmartGrid analysis is smaller by a factor of ten. The SmartGrid contains 15 analysis components, one of which consolidates the dependencies on the metamodel.

2.4.1. SmartGrid Overview

The Smart Grid Resilience Framework allows modelling and analyse critical infrastructures. With metamodel of this analysis, the topology of a smart grid can be modelled. The analysis allows for simulating cyberattacks; it also allows for determining the impact of such attacks on the infrastructure. These simulations can be coupled with a power load simulation and a simulation of critical infrastructures, which are developed by our research partners. In contrast to the previous case studies, the metamodel is integrated into the analysis.

2.4.2. SmartGrid Refactoring

Before we could refactor the SmartGrid analysis, we had to adapt the dependencies of the analysis code to the modular SmartGrid metamodel [6, 15]. The dependencies of the SmartGrid regarding the modular SmartGrid metamodel are like the structure shown in fig. 2.1. The dependencies on the modular SmartGrid metamodel are consolidated in the *smartgrid.attackersimulation* and the *smartgrid.impactanalysis* module. Although technically, these two modules represent two different analyses, we consider them as one. Each represents a analysis feature of the SmartGrid analysis. As we did not refactor the whole analysis, details regarding the refactoring will be presented in chapter 5.

3. Refactorings

In this chapter, we present the refactorings that analysis developers can use to apply our reference architecture for model-based analyses. We split the refactorings in *analysis class refactorings* and *analysis component refactorings*. Class refactorings are intended for the refactoring of classes of an analysis component and component refactorings are intended for the refactoring of analysis components. The split and merge of classes, the breaking of dependency cycles, and the inversion of dependencies are part of class refactorings. The component split refactorings (i. e., vertically and horizontally), and component merges are part of the analysis component refactorings. To apply the structure of the DSML, the following refactorings can be used. Figure 3.1 shows the legend for the figures used in this chapter.

3.1. Analysis class refactorings

Class refactorings are the foundation for the decomposition of the model-based analysis and adapt it to the structure of the corresponding DSML. It is not always necessary to apply all in this section presented refactorings to reach the desired result. These class refactorings equip the user with a set of refactoring operations to break the monolithic structure of model-based analysis and make it modular without changing existing behaviour. We distinguish four types of analysis class level refactorings: splitting a class, merging a class, breaking dependency cycles, and reversing dependencies.

3.1.1. Class split

Splitting a class is a typical refactoring operation where class elements, such as attributes and methods, are extracted and transferred into one or more new classes [4]. In language- and object-oriented design, the goal of the class split refactoring is to separate different concerns into separate classes to improve the comprehensibility of individual classes. The refactoring operation class split is shown in fig. 3.2. The class C has dependencies on the two language components $L1$ and $L2$. Our approach assumes that the underlying language is already modularised and partitioned. Therefore, if possible, a class should be split with more than one language component as a dependency. Additionally, whether the language

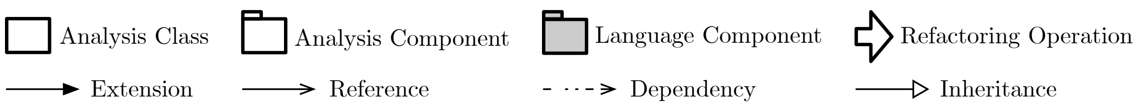


Figure 3.1.: Legend for the notational elements used to depict the refactoring operations

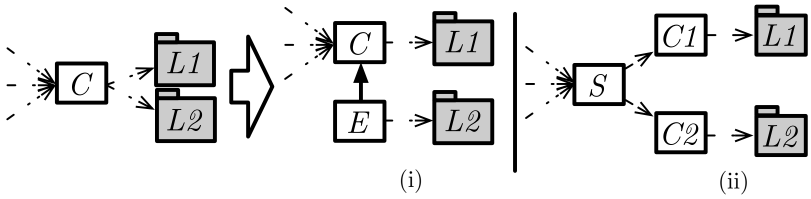


Figure 3.2.: Class split

components are at one layer or distributed over several layers in the architecture must be distinguished. Namely, when a class is split according to the structure of the language components, the refactored classes must be distributed according to their dependencies in the same architecture layers. It is shown in fig. 3.2(i) class C is extended by a new class E. Also, E takes properties of C; for this, the required properties are factored out from C to E. Incoming dependencies remain on C.

From a purely syntactical view, attributes, methods, references, containments, and inheritance can be factored out on the class level without complications. In the case of model-based analysis, it is often not possible to split a class according to the language structure. An analysis feature might need different language features to perform an analysis, but the structure of the DSML requires, that the analysis feature has no dependency to the language feature i. e., has no knowledge about the language feature. However, given the structure of the language, it is not always possible to separate a class as demanded by the reference architecture of the metamodel. This can occur if, for example, language components from different layers are used with dependencies on each other. Besides the elements that can be cleanly separated from a class and the components that do not have dependencies on the language component, we propose encapsulating the inseparable elements in a class and then placing them in the most specific layer. As it is shown in fig. 3.2(ii), instead of an extension class E, a specialisation class S is introduced and the incoming dependencies are shifted to S. In the worst-case scenario, the classes cannot be fully split, so that S holds dependencies of $L1$ and $L2$.

3.1.2. Class merge

Like the class split, the class merge is also a refactoring operation that originates in object-oriented design [4]. A class merge transfers attributes and methods of a class to another already existing class. The class merge is intended to consolidate concerns that

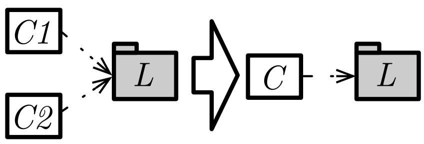


Figure 3.3.: Class merge

are distributed across classes. The class merge refactoring is shown in fig. 3.3. When a language component of the DSML is scattered, i. e., types of a language component are referenced by multiple classes and levels, the class merge can be used to merge these dependencies. This operation is applied by extracting attributes and methods of one class and then inserting them into another class. The result is an extended target class with attributes and behaviour of the source class. $C1$ and $C2$ have dependencies on the same language component L; the merge combines $C1$ and $C2$ into one new class, C which as a result, shares the dependencies on the desired language component L.

3.1.3. Breaking dependency cycles

Model-based analysis modelled according to our reference architecture must be cycle free. If the bad smell *cyclic dependencies*, known from object-oriented design, occurs the following refactoring operations show, how developers can break such cycles. Dependency cycles

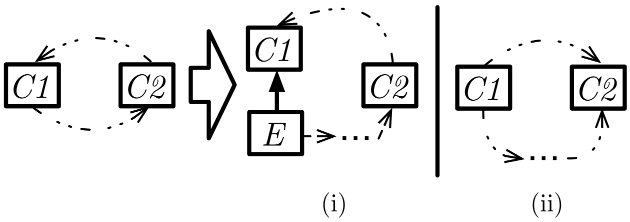


Figure 3.4.: Breaking dependency cycles

prevent easy extension of software systems [13], and according to Fowler [3], dependency cycles make a system harder to understand, thus, harder to maintain. The refactoring of breaking dependency cycles is shown in fig. 3.4 We assume, that the DSML does not contain any dependency cycles [6], and thus, the model-based analysis should also not contain any dependency cycles. As in language- and object-oriented design, we distinguish two refactoring operations to break dependency cycles. On the one hand, the previously presented class split can be used; on the other hand, the dependency inversion is also a valid option to break dependency cycles. The initial state is, that $C1$ and $C2$ depend on each other. The outgoing dependency of $C1$ is factored out into E if they contributed to the cycle. As a result, $C1$ is split, and $C1$ has no dependency on E; thus, the cycle no longer exists see fig. 3.4(i). The dependency inversion is described in the following section. Dependency inversion is one technique to tackle dependency cycles, as exemplified in fig. 3.4(ii).

3.1.4. Dependency inversion

According to Martin [11], abstractions (A) must not depend on specifics (S); instead, specifics must depend on abstractions. This statement is known as the dependency inversion principle. It originated in the object-oriented design and was later adapted to suit the design of DSMLs [6]. To tackle the problem when dependencies violate the reference architecture constrains, we present a refactoring solution that transfers the reference architecture for DSMLs to model-based analyses. The refactoring operations for dependency inversion are illustrated in fig. 3.5.

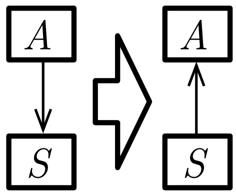


Figure 3.5.: Dependency inversion

If A is a specialisation of S, the inheritance is wrong and must be inverted; occurrences of S and A in the analysis also must be switched fig. 3.6(i). The inheritance is removed

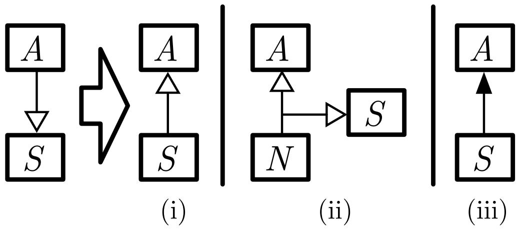


Figure 3.6.: Dependency inversion – Inheritance

if an atomic analysis feature is implemented by A and S. The new subclass of A and S, N, is introduced fig. 3.6(ii). Dependencies must be redirected to either A, S, or N; for this, incoming dependencies of A and S are used. If S is not a specialisation of A but a first-class analysis feature, the inheritance is removed and replaced by a reference from S to A fig. 3.6(iii).

A reference fig. 3.7 can be inverted using a class split fig. 3.7(i). When inverting the reference, a new class E is introduced. E replaces the reference from A to S. This option

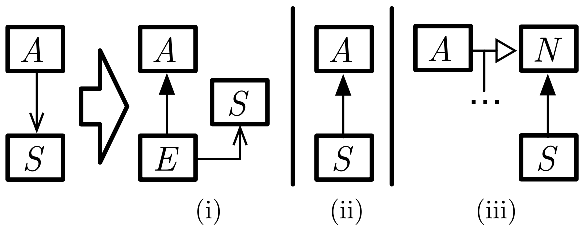


Figure 3.7.: Dependency inversion – Reference

should be chosen if S is a first-class analysis feature (i. e., an instance of S is not dependent on an instance of A). When numerous other classes refer to an instance of S, this is an indication. If S is a second-class analysis feature, which extends the functionality of A but is no further extended, a simple extends relation can be implemented fig. 3.7(ii). However, if S needs to be further specialised, introducing a common superclass N is advised fig. 3.7(iii). A bidirectional reference between two classes A and S is the simplest form of a dependency cycle (see fig. 3.4, and a special case of fig. 3.7). The bidirectional

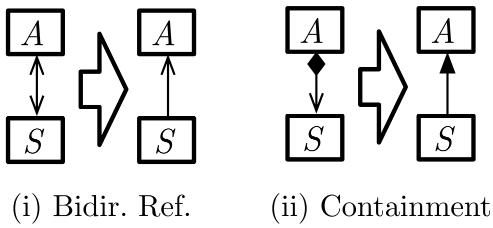


Figure 3.8.: Dependency inversion – Bidirectional Reference and Containment

nature implies a redundant reference, which can be removed so that only one reference remains, see fig. 3.8(i). Containment references can be removed by extracting an extension class S representing the desired feature. That way, features can be strictly separated, see fig. 3.8(ii).

3.2. Analysis component refactorings

Our approach uses several refactorings to adjust analysis components, dependencies, and classes. Many of these refactorings perform a split of an analysis component, which *RefactorLizar* supports. When splitting an analysis component, the analysis architect first selects the analysis component which needs to be split and then selects the corresponding language. *RefactorLizar* then automatically refactors the component accordingly to the corresponding language. If the analysis architect wants to perform a specific refactoring operation, *RefactorLizar* also supports manual evocation of all class level and component level refactorings.

3.2.1. Horizontal split

An analysis component must be split horizontally by the analysis architect if parts of an analysis component can be used independently of each other (cf. Single Responsibility Principle [10]). An initial indicator to split an analysis component is when an analysis component has dependencies on multiple language components. fig. 3.9(i) shows the potential best-case outcome; the components are unrelated. In fig. 3.9(ii), one of the analysis components is dependent on the other. In fig. 3.9(iii), the potential worst case is shown. The new components M and N may still share the original component's common part P. The parenthesis around P indicates that this component does not necessarily exist. All the analysis components may be mutually dependent. The dependencies of M and N must be adjusted according to the dependencies of the analysis feature they implement. The adjustment of the dependencies must be done by the analysis architect and the analysis component developer, in fig. 3.9(iv) the components M and N dependent on a common component P. The common analysis component of P also indicates an additional feature, which is an addition to the analysis feature graph.

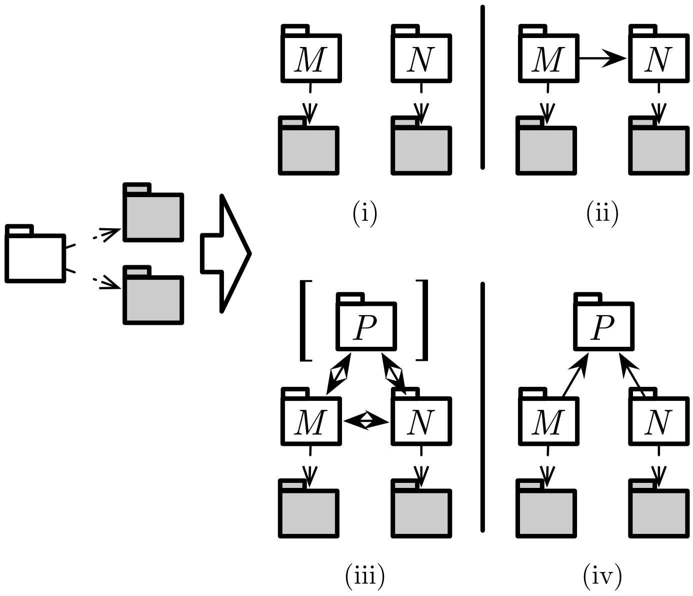


Figure 3.9.: Horizontal split

3.2.2. Vertical split

The vertical split is illustrated in fig. 3.10. The analysis architect performs this refactoring if

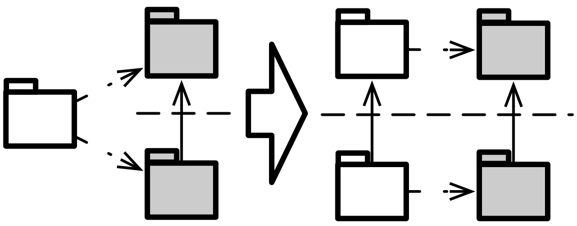


Figure 3.10.: Vertical split

the layer an analysis component could be assigned to is not clear. An indicator to vertically split an analysis component is when said component has dependencies on language components on different layers. A horizontal split is recommended if the language components are on the same layer. The analysis architect divides the analysis component so that each resulting analysis component can be assigned to one layer. The analysis component developer must split classes if necessary. After the refactoring, each resulting analysis component is assigned to its layer by the analysis architect. The resulting architecture could have dependencies that point from an abstract to a more specific layer. If this is the case, the analysis component developer must perform dependency inversion.

3.2.3. Merge

A merge refactoring could be advisable when more than one analysis component depends on the same language component and if the analysis components are located on the same layer. The analysis developer checks whether the dependent language features have a

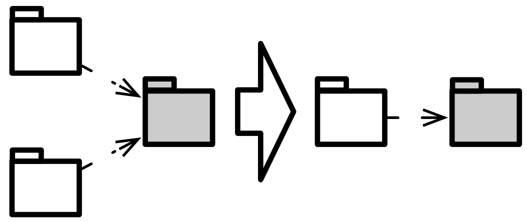


Figure 3.11.: Merge

mandatory feature relation or if the analysis components form a dependency cycle. If one of these constructs can be found in the architecture, the analysis architect should consider merging those features and their analysis components fig. 3.11.

3.2.4. Extension extraction

The analysis architect uses extension extraction refactoring if an analysis component contain content that does not belong to the feature it implements. An indicator for refac-

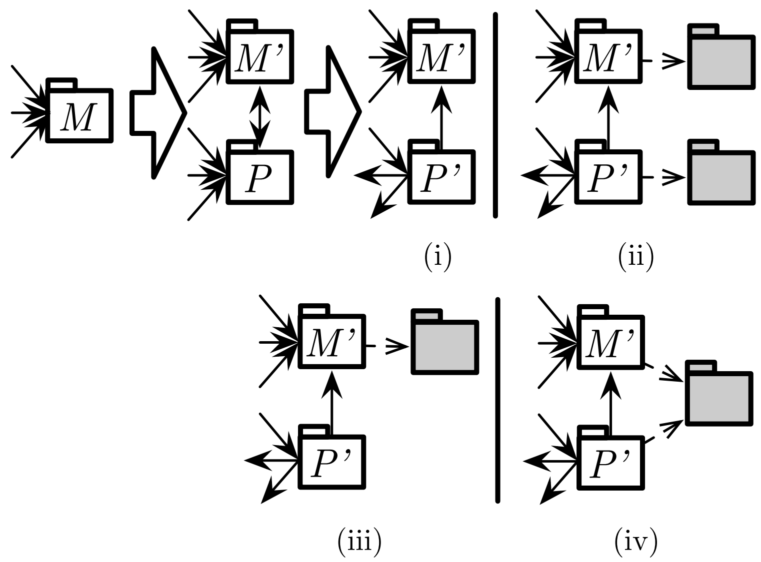


Figure 3.12.: Extension extraction

toring is if the optional content cannot be used independently. The extension extraction refactoring is depicted in fig. 3.12 – the analysis architect factors out the optional content of M into a new analysis component P. The remainder of M is denoted as M'. The classes of component M must be split if they should be located in P but contain optional properties that belong to M'. The analysis component developer also does this refactoring. If a class has dependencies on multiple language components, which cannot be factored out, the class must be put in the most specialised analysis component. The following step reverses all dependencies from elements of M' to P. Incoming dependencies on P must be considered for dependency inversion. The result of the dependency inversion is shown as outgoing dependencies of P'. The refactoring can be performed if the analysis components have no dependencies on any language component. However, if M has dependencies on multiple language components, each dependency should be refactored into one dedicated analysis component (see fig. 3.12(ii)). If the optional content of P' represents a dedicated analysis component that has no representation in the language, P' must be refactored into a dedicated analysis component with no dependencies on the language (see fig. 3.12(iii)). If it is reasonable to separate optional content, P' but the dependencies on one language component cannot be separated fig. 3.12(iv) must be applied.

3.2.5. Feature support extraction

The refactoring feature support extraction is still a form of extension extraction refactoring [6]. The refactoring is depicted in fig. 3.13. It has the same impact as separating P

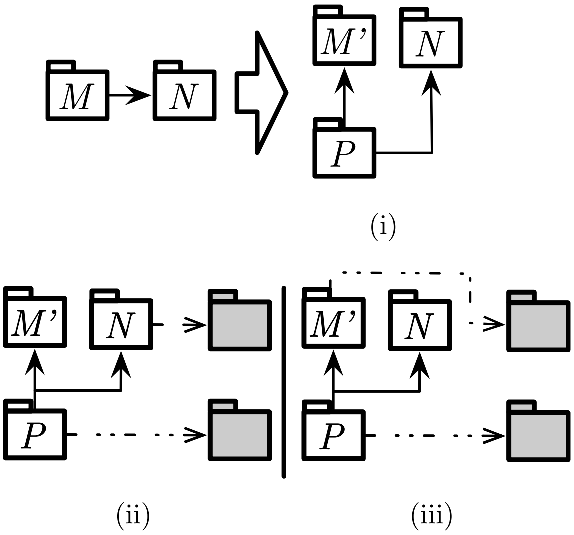


Figure 3.13.: Feature support extraction

shown in fig. 3.12. The analysis architect performs the feature support extraction if a part P of an analysis component M is dependent on another analysis component N, and M cannot be used without N. The analysis architect separates P into its analysis component. The remainder of M is denoted as M'. P is dependent on M', and N. Dependencies must be inversed if M' has dependencies to P. To separate the content of both analysis features, the analysis component developer performs class split refactorings. P, the extension of M' contains content of N, P adds support for N to M'. Thus, it is referred to as feature support extraction.

4. Evaluation Tooling

We have developed a Java library to support the evaluation of our case studies. Besides the evaluation, the library also supports the analysis and refactoring of model-based analyses. The library supports Java- and EMF-based analyses. The library, called RefactorLizar[1], serves three purposes. Determining cohesion, coupling, and complexity of model-based analyses, see section 4.1. Analysis of model-based analyses regarding reference architecture violations, see section 4.2. Automated refactoring of model-based analysis, see section 4.3.

4.1. RefactorLizar – Evaluation Library

RefactorLizar allows developers to determine cohesion, coupling, and complexity of model-based analyses using the hypergraph metrics of Allen et al. [1]. The evaluation of meta-models is not supported by RefactorLizar Evaluation Library.

4.2. RefactorLizar – Analysis Library

The analysis part of RefactorLizar provides information about a modularized model-based analysis. In order for the analysis to function, the developer must provide a DSML and a corresponding model-based analysis as input. RefactorLizar consists of the following four analyses: 4.2.1 *Feature Scatter identification*, 4.2.2 *Language Blob identification*, 4.2.3 *Identification of layer violations*, and 4.2.4 *Identification of dependency cycles*.

4.2.1. Feature Scatter identification

When multiple analysis components have dependencies on the same DSML language type, we define it as Feature Scatter. A Feature Scatter violates our reference architecture, as an analysis component should only depend on one language feature. Besides the multiple dependencies, the feature and its corresponding component must be located on a single layer. RefactorLizar can identify the scattering of features; it provides the developer with a list of components that depend on a single feature.

4.2.2. Language Blob identification

When analysis components have multiple dependencies on DSML language types, we define it as Language Blob. A Language Blob violates our reference architecture, as an analysis component should have only one DSML feature as dependency. RefactorLizar

[1] https://github.com/MoSimEngine/RefactorLizar

supports the developer to identify components that have dependencies to more than one DSML feature. The analysis result provides the developer with a list of language features, a component depends on. The developer can define a threshold to set the minimal amount of dependencies before a component is added to the list.

4.2.3. Identification of layer violations

When a model-based analysis feature is on a different layer than its corresponding model-based analysis component, we define it as a layer violation. Also, when a model-based analysis feature is on a different layer than its corresponding DSML feature, we define it as a layer violation. RefactorLizar allows the developer to detect when dependencies between layers point in the wrong direction or surpass adjacent layers. The layer identification of analysis components requires further annotation by the analysis developer, while the layer identification of referenced DSML types is made automatically.

4.2.4. Identification of dependency cycles

A bidirectional dependency between analysis components is the simplest form of a dependency cycle. RefactorLizar can detect dependency cycles on class and component level.

4.3. RefactorLizar – Refactoring Library

RefactorLizar supports the following basic refactorings: Move type members, introduce inheritance, adapt interface extension, change the visibility of methods and attributes, delete classes, delete methods and attributes, and create new types. RefactorLizar can provide these refactorings automatically: Class Split, Class Merge, Breaking Dependency Cycle, Dependency Inversion, Horizontal Split, Vertical Split, Merge, and Extension Extraction.

4.4. RefactorLizar – Reference implementation

This project provides a command-line interface for the RefactorLizar library. We made a reference implementation to demonstrate the analysis features of RefactorLizar. The reference implementation is available on our GitHub page[2] and supplementary material [9]. The implementation is provided as a *Command Line Interface* (CLI) tool. We also plan to provide a visual interface for the RefactorLizar library as a Visual Studio Code extension.

4.4.1. Commands

RefactorLizarCLI utilizes PicoCLI[3] and GraalVM[4] thus, commands can be started via the provided binary or via `gradle run -args="<command/s>"`.

[2] https://github.com/MoSimEngine/RefactorLizarCLI
[3] https://picocli.info/
[4] https://www.graalvm.org/

4.4.1.1. evaluateCode

The command *evaluateCode* evaluates hypergraph code metrics for the given source path . The arguments `data-types` and `observed-system` is the path to file for ignored/included types. Every line in this file is seen as a regex tested against the qualified type names. The `data-types` parameter represents the ignored types and the `observed-system` parameter represents the included types. The `code` argument provides the path to the analysis to evaluate.

4.4.1.2. adaptDependencies

The command *adaptDependencies* changes imports of simulator code according to the new, modular metamodel. The command requires a `CSV` file that contains the mapping of the modular metamodel types mapped to the monolithic metamodel types. The argument `csv-path` provides the path to the CSV file. The argument `simulator-code` provides the path to the analysis.

4.4.1.3. findDependencyCycleSmell

The command *findDependencyCycleSmell* finds occurrences of the dependency cycle smell. The `analysis-level` argument sets the detail level of the result. Available analysis levels are type, component and package. The `language` argument is the path to the metamodel. The `simulator` argument is the path to the analysis code. If the code is eclipse-based the flag `input-type-eclipse` allows to handle eclipse-based analyses.

4.4.1.4. findDependencyDirectionSmell

Find occurrences of the dependency direction smell. Layers must be ordered from bottom to top and separated by ';'. Available analysis levels are type, component and package. The `analysis-level` argument sets the detail level of the result. Available analysis levels are type, component and package. The `language` argument is the path to the metamodel. The `simulator` argument is the path to the analysis code.

4.4.1.5. showTypesInMetamodels

The command *showTypesInMetamodels* lists all metamodel types. The argument `language-root` points to the root of the metamodel. The result can be used to determine the utilisation of a metamodel in a model-based analysis. The utilisation is the number of all types in a metamodel in relation to the types of a metamodel used in a model-based analysis.

4.4.1.6. findFeatureScatteringSmell

Find occurrences of the feature scattering smell. The `analysis-level` argument sets the detail level of the result. Available analysis levels are type, component and package. The `language` argument is the path to the metamodel. The `simulator` argument is the path to the analysis code. If the code is eclipse-based the flag `input-type-eclipse` allows to handle eclipse-based analyses.

4.4.1.7. findDependencyLayerSmell

Find occurrences of the improper simulator layering smell. Available analysis levels are type, component and package. The `analysis-level` argument sets the detail level of the result. Available analysis levels are type, component and package. The `language` argument is the path to the metamodel. The `simulator` argument is the path to the analysis code. If the code is eclipse-based the flag `input-type-eclipse` allows to handle eclipse-based analyses.

4.4.1.8. findLanguageBlobSmell

Find occurrences of the language blobs smell. Available analysis levels are type, component and package. The `analysis-level` argument sets the detail level of the result. Available analysis levels are type, component and package. The `language` argument is the path to the metamodel. The `simulator` argument is the path to the analysis code. If the code is eclipse-based the flag `input-type-eclipse` allows to handle eclipse-based analyses.

5. Evaluation Data

5.1. Evolution Scenarios

In this section, we present the evolution scenarios of our four case studies. We identified ten scenarios per case study, ergo 40 scenarios in total. Each case study provided historical evolution scenarios; we did not have to define potential or random evolution scenarios. Historical evolution scenarios can affect files without dependencies on the DSML; thus, we did not apply any refactoring to these files. Also, we did not consider these files when we calculated the metrics cohesion, coupling, and complexity. For each case study, we provide sources for the DSML and model-based analysis in their monolithic and modular state. The scenarios can also be found in our reproduction package [9]. To correctly identify the scenario in the source code, we provide a commit hash or revision number and the date when the commit occurred.

5.1.1. SimuLizar

For the model-based analysis SimuLizar, we identified ten historical evolution scenarios. The reproduction data for SimuLizar contains the refactored code of the model-based analysis. Table 5.1 contains links to the monolithic and modular model-based analysis.

	Name	Source	Branch
Language	PCM	[15]	–
Modular Language	mPCM	[15]	–
Analysis	SimuLizar	Palladio-Analyzer-SimuLizar[1]	master: b6b69b4f1
Modular Analysis	mSimuLizar	mSimuLizar [9]	–

Table 5.1.: Overview SimuLizar Projects

5.1.1.1. Scenario 01 – RepositoryComponentSwitch uses Extensible RDSeffSwitches

The first scenario is the commit 7542134. The commit occurred on Monday, April 24th 2017. In the monolith, four files are changed. The following files are affected by the commit: RDSeffSwitch, RepositoryComponentSwitch, AbstractRDSeffSwitchFactory, and IComposableSwitch.

[1] `https://github.com/PalladioSimulator/Palladio-Analyzer-SimuLizar`

5.1.1.2. Scenario 02 – Deleted ModelAccess Class

The second scenario is the commit 534d5521. The commit occurred on Friday, August 17th 2018. In the monolith, 28 files are changed. The following files are affected by the commit: IModelAccess, ModelAccess, ModelAccessUseOriginalReferences, InterpreterDefaultContext, AbstractProbeFrameworkListener, ProbeFrameworkListener, EvaluateResultsJob, PCMStartInterpretationJob, AbstractResourceEnvironmentObserver, AbstractUsageEvolutionObserver, AbstractUsageModelObserver, ResourceEnvironmentSyncer, AbstractReconfigurationLoader, AbstractReconfigurator, AbstractReconfigurator, IReconfigurationEngine, IReconfigurationLoader, ReconfigurationProcess, Reconfigurator, AbstractSimuLizarRuntimeState, SimuLizarRuntimeState, SimulatedBasicComponentInstance, PeriodicallyTriggeredUsageEvolver, SimulatedUsageModels, UsageEvolverFacade, FileUtil, PCMPartitionManager, and ResourceUtil.

5.1.1.3. Scenario 03 – Fix Project Structure - Migrate RDSeffSwitch to Tycho

The third scenario is the commit 02511a37. The commit occurred on Monday, July 30th 2018. In the monolith, three files are changed. The following files are affected by the commit: AbstractRDSeffSwitchFactory, ExplicitDispatchComposedSwitch, and IComposableSwitch.

5.1.1.4. Scenario 04 – Added Mechanism to Explicitly Switch Based on Superclass

The fourth scenario is the commit d973511. The commit occurred on Tuesday, December 12th 2017. In the monolith, three files are changed. The following files are affected by the commit: RepositoryComponentSwitch, AbstractRDSeffSwitchFactory, and ExplicitDispatchComposedSwitch.

5.1.1.5. Scenario 05 – Add Monitorrepository to Feature Dependencies

The fifth scenario is the revision r34181. The commit occurred on Monday, April 24th 2017. In the monolith, four files are changed. The following files are affected by the commit: AbstractRDSeffSwitchFactory, IComposableSwitch, RDSeffSwitch, and RepositoryComponentSwitch.

5.1.1.6. Scenario 06 – Fixed Metadata for the HDD Patch

The sixth scenario is the revision r33820. The commit occurred on Friday, November 11th 2016. In the monolith, one files are changed. The following file is affected by the commit: RDSeffSwitch.

5.1.1.7. Scenario 07 – Include New Aggregation Plugin into Simulizar Feature

The seventh scenario is the commit r32804. The commit occurred on Friday, August 5th 2016. In the monolith, six files are changed. The following files are affected by the commit: AbstractProbeFrameworkListener, PRMRecorder, AbstractModelObserver, ResourceEnvironmentSyncer, AbstractSimuLizarRuntimeState, and MonitorRepositoryUtil.

5.1.1.8. Scenario 08 – Only Record Runtime Measurements

The eighth scenario is the revision r32416. The commit occurred on Wednesday, July 6th 2017. In the monolith, 23 files are changed. The following files are affected by the commit: InterpreterDefaultContext, AbstractProbeFrameworkListener, AbstractRecordingProbeFrameworkListenerDecorator, ProbeFrameworkListener, PRMRecorder, AbstractModelObserver, AbstractResourceEnvironmentObserver, AbstractUsageEvolutionObserver, AbstractUsageModelObserver, IModelObserver, ResourceEnvironmentSyncer, AbstractSimuLizarRuntimeState, IRuntimeStateAccessor, SimuLizarRuntimeState, SimuLizarRuntimeStateAbstract, SimulatedComponentInstance, SimulatedCompositeComponentInstance, LoopingUsageEvolver, PeriodicallyTriggeredUsageEvolver, StretchedUsageEvolver, UsageEvolverFacade, FileUtil, and MonitorRepositoryUtil.

5.1.1.9. Scenario 09 – Generalized Response Times Aggregator

The ninth scenario is the revision r32166. The commit occurred on Tuesday, Mai 31st 2016. In the monolith, four files are changed. The following files are affected by the commit: AbstractSimuLizarRuntimeState, ComponentInstanceRegistry, SimulatedBasicComponentInstance, and SimulatedComponentInstance.

5.1.1.10. Scenario 10 – Added Missing Reconfiguration Rule

The tenth scenario is the revision r31800. The commit occurred on Tuesday, April 19th 2016. In the monolith, six files are changed. The following files are affected by the commit: EventNotificationHelper, RepositoryComponentSwitch, AbstractInterpreterListener, AbstractRecordingProbeFrameworkListenerDecorator, AssemblyProvidedOperationPassedEvent, and IInterpreterListener.

5.1.2. Camunda

For the model-based analysis Camunda, we identified ten historical evolution scenarios. The reproduction data for Camunda contains the classes of the ten scenarios. Each scenario is divided into two folders, the classes of the monolithic version is contained in the folder *before* and the classes of the modular version after the refactoring is contained in the *after* folder. Table 5.2 contains links to the monolithic and modular model-based analysis.

	Name	**Source**	**Branch**
Language	BPMN/Camunda	[15]	–
Modular Language	mBPMN/Camunda	[15]	–
Analysis	Camunda	Camunda GitHub[2]	master:f5c2d559d
Modular Analysis	mCamunda	mCamunda [9]	–

Table 5.2.: Overview Camunda Projects

5.1.2.1. Scenario 01 – Add Timeout Task Listener

The first scenario is the commit d53583a. The commit occurred on Wednesday, August 21st 2019. In the monolith, five files are changed. See fig. A.1 and fig. A.2. The following files are affected by the commit: AbstractBaseElementBuilder, AbstractCatchEventBuilder, AbstractUserTaskBuilder, CamundaTaskListenerImpl, and CamundaTaskListener.

5.1.2.2. Scenario 02 – Introduce errorMessage for Error Definitions

The second scenario is the commit b129522. The commit occurred on Friday, July 5th 2019. In the monolith, six files are changed. See fig. A.3 and fig. A.4. The following files are affected by the commit: AbstractBaseElementBuilder, AbstractBoundaryEventBuilder, AbstractEndEventBuilder, AbstractErrorEventDefinitionBuilder, AbstractStartEventBuilder, and BpmnModelConstants.

5.1.2.3. Scenario 03 – Add Variable Specification to Conditional Event

The third scenario is the commit 14ad97ae. The commit occurred on Wednesday, October 5th 2016. In the monolith, four files are changed. See fig. A.5 and fig. A.6. The following files are affected by the commit: AbstractConditionalEventDefinitionBuilder, BpmnModelConstants, ConditionalEventDefinitionImpl, and ConditionalEventDefinition.

5.1.2.4. Scenario 04 – Remove incrementalIntervals Property

The fourth scenario is the commit a337b8f6. The commit occurred on Friday, September 8th 2017. In the monolith, four files are changed. See fig. A.7 and fig. A.8. The following files are affected by the commit: Bpmn, AbstractFlowNodeBuilder, BpmnModelConstants, CamundaIncrementalIntervalsImpl, and CamundaIncrementalIntervals.

[2] `https://github.com/MoSimEngine/camunda-bpm-platform`

5.1.2.5. Scenario 05 – Set Marker to Visible for Exclusive Gateway

The fifth scenario is the commit 7cf3cdff. The commit occurred on Thursday, June 1st 2017. In the monolith, one file is changed. See fig. A.9 and fig. A.10. The following file is affected by the commit: AbstractFlowNodeBuilder.

5.1.2.6. Scenario 06 – Removed errorMessage Attribute in endErrorEvent

The sixth scenario is the commit 4a5d7bc7c. The commit occurred on Monday, June 6th 2016. In the monolith, eight files are changed. See fig. A.11 and fig. A.12. The following files are affected by the commit: AbstractBaseElementBuilder, AbstractBoundaryEventBuilder, AbstractEndEventBuilder, AbstractErrorEventDefinitionBuilder, AbstractStartEventBuilder, BpmnModelConstants, ErrorImpl, and Error.

5.1.2.7. Scenario 07 – Added Error Definition Variables

The seventh scenario is the commit 31e9a1324. The commit occurred on Thursday, June 2nd 2016. In the monolith, eleven files are changed. See fig. A.13 and fig. A.14. The following files are affected by the commit: AbstractBaseElementBuilder, AbstractBoundaryEventBuilder, AbstractEndEventBuilder, AbstractErrorEventDefinitionBuilder, AbstractStartEventBuilder, ErrorEventDefinitionBuilder, BpmnModelConstants, ErrorEventDefinitionImpl, ErrorImpl, Error, and ErrorEventDefinition.

5.1.2.8. Scenario 08 – Add Convenience Methods to Allow Using Classes Instead

The eighth scenario is the commit 1d2a508c. The commit occurred on Friday, March 24th 2017. In the monolith, six files are changed. See fig. A.15 and fig. A.16. The following files are affected by the commit: AbstractBusinessRuleTaskBuilder, AbstractCallActivityBuilder, AbstractFlowNodeBuilder, AbstractSendTaskBuilder, AbstractServiceTaskBuilder, and AbstractUserTaskBuilder.

5.1.2.9. Scenario 09 – Create and Reference Message with the Fluent Builder

The ninth scenario is the commit 677b3c6. The commit occurred on Monday, February 1st 2016. In the monolith, six files are changed. See fig. A.17 and fig. A.18. The following files are affected by the commit: AbstractBaseElementBuilder, AbstractCatchEventBuilder, AbstractFlowNodeBuilder, AbstractReceiveTaskBuilder, AbstractSendTaskBuilder, and AbstractThrowEventBuilder.

5.1.2.10. Scenario 10 – Add Support for camunda:connector Extension Element

The tenth scenario is the commit c30dbc8e. The commit occurred on Tuesday, August 5th 2014. In the monolith, six files are changed. See fig. A.19 and fig. A.20. The following files are affected by the commit: Bpmn, BpmnModelConstants, CamundaConnectorIdImpl, CamundaConnectorImpl, CamundaConnector, and CamundaConnectorId.

5.1.3. KAMP4aPS

For the model-based analysis KAMP4aPS, we identified ten historical evolution scenarios. The reproduction data for KAMP4aPS contains the classes of the ten scenarios. Each scenario is divided into two folders, the classes of the monolithic version is contained in the folder *before* and the classes of the modular version after the refactoring is contained in the *after* folder. Table 5.3 contains links to the monolithic and modular model-based analysis.

	Name	**Source**	**Branch**
Language	KAMP4aPS Lang	[15]	–
Modular Language	mKAMP4aPS Lang	[15]	–
Analysis	KAMP4aPS	KAMP4aPS GitHub[3]	master: HEAD
Modular Analysis	mKAMP4aPS	mKAMP4aPS [9]	–

Table 5.3.: Overview KAMP4aPS Projects

5.1.3.1. Scenario 01 – Add Lookup for Interface Elements

The scenario is the commit 3126580b. The commit occurred on Sunday, March 19th 2017. In the monolith, four files are changed. See fig. A.21 and fig. A.22. The following files are affected by the commit: ArchitectureAnnotationLookup, AbstractKAPSDifferenceCalculation, AbstractKAPSEnrichedWorkplanDerivation, and SwitchChanges.

5.1.3.2. Scenario 02 – Add Super Type to Mechanical Assembly

The scenario is the commit 2d37dc02. The commit occurred on Monday, October 23rd 2017. In the monolith, four files are changed. See fig. A.23 and fig. A.24. The following files are affected by the commit: APSArchitectureModelLookup, ModuleChanges, MicroSwitchModuleChange, and RampChange.

5.1.3.3. Scenario 03 – Add Class for Micro Switch Change

The scenario is the commit c17f986e5. The commit occurred on Friday, August 18th 2017. In the monolith, six files are changed. See fig. A.25 and fig. A.26. The following files are affected by the commit: APSArchitectureModelLookup, APSChangePropagationAnalysis, APSSubactivityDerivation, MicroSwitchModuleChange, SwitchChanges, and LabelCustomizing.

5.1.3.4. Scenario 04 – Add Meta Class for Change

The scenario is the commit 1f78d0c0. The commit occurred on Friday, August 18th 2017. In the monolith, ten files are changed. See fig. A.27 and fig. A.28. The following files are affected by the commit: Change, ComponentChanges, InterfaceChanges, ModuleChanges,

[3] `https://github.com/KAMP-Research/KAMP4APS`

StructureChanges, BusChanges, RampChange, SensorChanges, SignalInterfacePropagation, and SwitchChanges.

5.1.3.5. Scenario 05 – Update Ramp Change Scenario

The scenario is the commit 3f5acd29. The commit occurred on Monday, May 14th 2018. In the monolith, two files are changed. See fig. A.29 and fig. A.30. The following files are affected by the commit: APSChangePropagationAnalysis, and RampChange.

5.1.3.6. Scenario 06 – Refactoring Names of Change Classes

The scenario is the commit 8491dd9b. The commit occurred on Tuesday, August 15th 2017. In the monolith, four files are changed. See fig. A.31 and fig. A.32. The following files are affected by the commit: BusChanges, SensorChanges, SignalInterfacePropagation, and SwitchChanges.

5.1.3.7. Scenario 07 – Introduce HMI

The scenario is the commit d54511fe. The commit occurred on Thursday, April 26th 2018. In the monolith, six files are changed. See fig. A.33 and fig. A.34. The following files are affected by the commit: APSArchitectureVersion, APSArchitectureVersionPersistency, APSChangePropagationAnalysis, and APSDifferenceCalculation.

5.1.3.8. Scenario 08 – Adapt Change Propagation Analysis Regarding PLC Entry Points

The scenario is the commit 5dae880b. The commit occurred on Tuesday, February 27th 2018. In the monolith, five files are changed. See fig. A.35 and fig. A.36. The following files are affected by the commit: APSArchitectureModelFactoryFacade, APSArchitectureVersion, APSArchitectureVersionPersistency, APSChangePropagationAnalysis, and InterfaceChanges.

5.1.3.9. Scenario 09 – Introduce Duplicate Removal

The scenario is the commit a5dcc00c. The commit occurred on Wednesday, January 11th 2017. In the monolith, five files are changed. See fig. A.37 and fig. A.38. The following files are affected by the commit: AbstractKAPSChangePropagationAnalysis, ArchitectureAnnotationLookup, ArchitectureModelLookup, ArchitectureVersion, and ArchitectureVersionPersistency.

5.1.3.10. Scenario 10 – Refactor Function Names and Introduce Version

The scenario is the commit 1d2a508c. The commit occurred on Wednesday, Januray 11th 2017. In the monolith, three files are changed. See fig. A.39 and fig. A.40. The following files are affected by the commit: AbstractKAPSChangePropagationAnalysis, ArchitectureModelLookup, and BusChanges.

5.1.4. SmartGrid

For the model-based analysis SmartGrid, we identified ten historical evolution scenarios. The reproduction data for SmartGrid contains the classes of the ten scenarios. Each scenario is divided into two folders, the classes of the monolithic version is contained in the folder *before* and the classes of the modular version after the refactoring is contained in the *after* folder. Table 5.4 contains links to the monolithic and modular model-based analysis.

	Name	Source	Branch
Language	SmartGridLang	[15]	–
Modular Language	mSmartGridLang	[15]	–
Analysis	SmartGrid	SmartGrid GitHub[4]	master: HEAD
Modular Analysis	mSmartGrid	mSmartGrid [9]	–

Table 5.4.: Overview SmartGrid Projects

5.1.4.1. Scenario 01 – Pass Data to Power Load Sim Properly

The scenario is the commit dfe199815. The commit occurred on Friday, November 24th 2017. In the monolith, one file is changed. See fig. A.41 and fig. A.42. The following file is affected by the commit: ReactiveSimulationController.

5.1.4.2. Scenario 02 – Added Report Generation for Attacker Simulation

The scenario is the commit c8280939. The commit occurred on Sunday, April 23th 2017. In the monolith, one file is changed. See fig. A.43 and fig. A.44. The following file is affected by the commit: ReportGenerator.

5.1.4.3. Scenario 03 – Fixed to Support String IDs

The scenario is the commit 72ecaa73. The commit occurred on Tuesday, October 17th 2017. In the monolith, two files are changed. See fig. A.45 and fig. A.46. The following files are affected by the commit: GraphAnalyzer, and Tarjan.

5.1.4.4. Scenario 04 – Added Init Methods with Maps as Parameter

The scenario is the commit 2d7a9c46. The commit occurred on Friday, February 7th 2020. In the monolith, eight files are changed. See fig. A.47 and fig. A.48. The following files are affected by the commit: LocalHacker, ViralHacker, HashMapHelper, GraphAnalyzer, IAttackerSimulation, IImpactAnalysis, ImpactAnalysisMock, and NoAttackerSimulation.

[4] `https://github.com/kit-sdq/Smart-Grid-ICT-Resilience-Framework`

5.1.4.5. Scenario 05 – Added rootNode Search Viral Hacker

The scenario is the commit 1648636e. The commit occurred on Friday, November 22nd 2019. In the monolith, five files are changed. See fig. A.49 and fig. A.50. The following files are affected by the commit: LocalHacker, ViralHacker, ScenarioModelHelper, ReactiveSimulationController, and TestClientRMI.

5.1.4.6. Scenario 06 – Finalizing the RCP Commands

The scenario is the commit aae4a894. The commit occurred on Monday, July 27th 2020. In the monolith, eleven files are changed. See fig. A.51 and fig. A.52. The following files are affected by the commit: FileSystemHelper, Activator, SmartgridRCPApplication, ControllerCommand, GetModifiedPowerspecsCommand, InitTopoCommand, SimControlCommands, EObjectsHelper, LocalController, ReactiveSimulationController, and RCPCall.

5.1.4.7. Scenario 07 – Local Controller Without a Launch Configuration

The scenario is the commit 63ae1f4. The commit occurred on Friday, February 7th 2020. In the monolith, four files are changed. See fig. A.53 and fig. A.54. The following files are affected by the commit: ITimeProgressor, NoOperationTimeProgressor, LocalController, and ReactiveSimulationController.

5.1.4.8. Scenario 08 – Nodes are Now Randomly Hacked When Using Full Meshed Hacking

The scenario is the commit 3d81da9e. The commit occurred on Friday, January 15th 2016. In the monolith, one file is changed. See fig. A.55 and fig. A.56. The following file is affected by the commit: ViralHacker.

5.1.4.9. Scenario 09 – Modified Attacker Simulation to Support Disabling Root Node for Virus

The scenario is the commit 5ee72f70. The commit occurred on Tuesday, December 15th 2015. In the monolith, two files are changed. See fig. A.57 and fig. A.58. The following files are affected by the commit: LocalHacker, and ViralHacker.

5.1.4.10. Scenario 10 – Added Boolean Method to Attacker Simulation that Indicates if Attributes Can be Used or Not

The scenario is the commit 4c257bea. The commit occurred on Friday, November 13th 2015. In the monolith, two files are changed. See fig. A.59 and fig. A.60. The following files are affected by the commit: LocalHacker, and ViralHacker.

Bibliography

[1] Edward B. Allen, Sampath Gottipati, and Rajiv Govindarajan. “Measuring size, complexity, and coupling of hypergraph abstractions of software: An information-theory approach”. English. In: *Software Quality Journal* 15.2 (2007), pp. 179–212.

[2] Kiana Busch et al. “A Metamodel-Based Approach to Calculate Maintainability Task Lists of PLC Programs for Factory Automation”. In: *44th Annual Conference of the IEEE Industrial Electronics Society (IECON)*. IEEE, 2018.

[3] Martin Fowler. “Reducing coupling”. In: *IEEE Software* 18.4 (2001), pp. 102–104.

[4] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999. ISBN: 0-201-48567-2.

[5] R. Heinrich, K. Busch, and S. Koch. “A Methodology for Domain-Spanning Change Impact Analysis”. In: *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. Los Alamitos, CA, USA: IEEE Computer Society, Aug. 2018, pp. 326–330. DOI: `10.1109/SEAA.2018.00060`. URL: `https://doi.ieeecomputersociety.org/10.1109/SEAA.2018.00060`.

[6] Robert Heinrich, Misha Strittmatter, and Ralf Heinrich Reussner. “A Layered Reference Architecture for Metamodels to Tailor Quality Modeling and Analysis”. In: *IEEE Transactions on Software Engineering* 47.4 (2019), pp. 775–800. ISSN: 0098-5589. DOI: `10.1109/TSE.2019.2903797`.

[7] Robert Heinrich et al. “Architecture-based Analysis of Changes in Information System Evolution”. In: *Softwaretechnik-Trends*. Vol. 35(2). 2015.

[8] Robert Heinrich et al. “Architecture-based change impact analysis in cross-disciplinary automated production systems”. In: *Journal of Systems and Software* 146 (2018), pp. 167–185. ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2018.08.058`.

[9] Sandro Koch, Robert Heinrich, and Ralf Reussner. *Supplementary Material to "A Layered Reference Architecture for Model-based Quality Analysis"*. Feb. 2022. DOI: `10.6084/m9.figshare.19228377.v1`. URL: `https://figshare.com/articles/dataset/Supplementary_Material_to_A_Layered_Reference_Architecture_for_Model-based_Quality_Analysis_/19228377`.

[10] R.C. Martin et al. *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt series. Pearson Education, 2003. ISBN: 9780135974445. URL: `https://books.google.de/books?id=0HYhAQAAIAAJ`.

[11] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.

[12] Philipp Merkle and Jörg Henss. "EventSim – An Event-driven Palladio Software Architecture Simulator". In: *Palladio Days 2011 Proceedings (appeared as technical report)*. Karlsruhe: KIT, Fakultät für Informatik, 2011, pp. 15–22.

[13] David Lorge Parnas. "Designing Software for Ease of Extension and Contraction". In: *Transactions on Software Engineering* 2 (1979), pp. 128–138.

[14] Kiana Rostami et al. "Architecture-based Change Impact Analysis in Information Systems and Business Processes". In: *2017 IEEE International Conference on Software Architecture (ICSA2017)*. IEEE, 2017, pp. 179–188.

[15] Misha Strittmatter, Robert Heinrich, and Ralf Reussner. *Supplementary Material for the Evaluation of the Layered Reference Architecture for Metamodels to Tailor Quality Modeling and Analysis*. Tech. rep. 11. Karlsruhe Institute of Technology, 2018. 42 pp. URL: http://dx.doi.org/10.5445/IR/1000089243.

A. Appendix

A.1. Camunda

Figure A.1.: Camunda scenario 01 - before refactoring

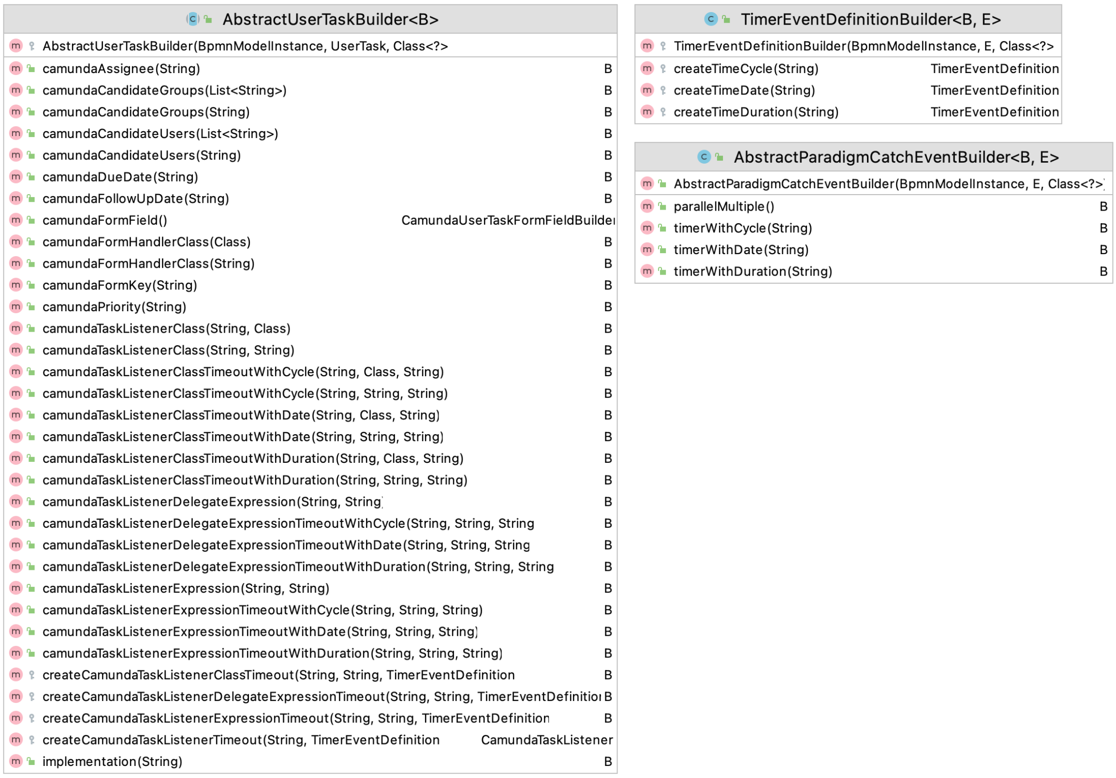


Figure A.2.: Camunda scenario 01 - after refactoring

Figure A.3.: Camunda scenario 02 - before refactoring

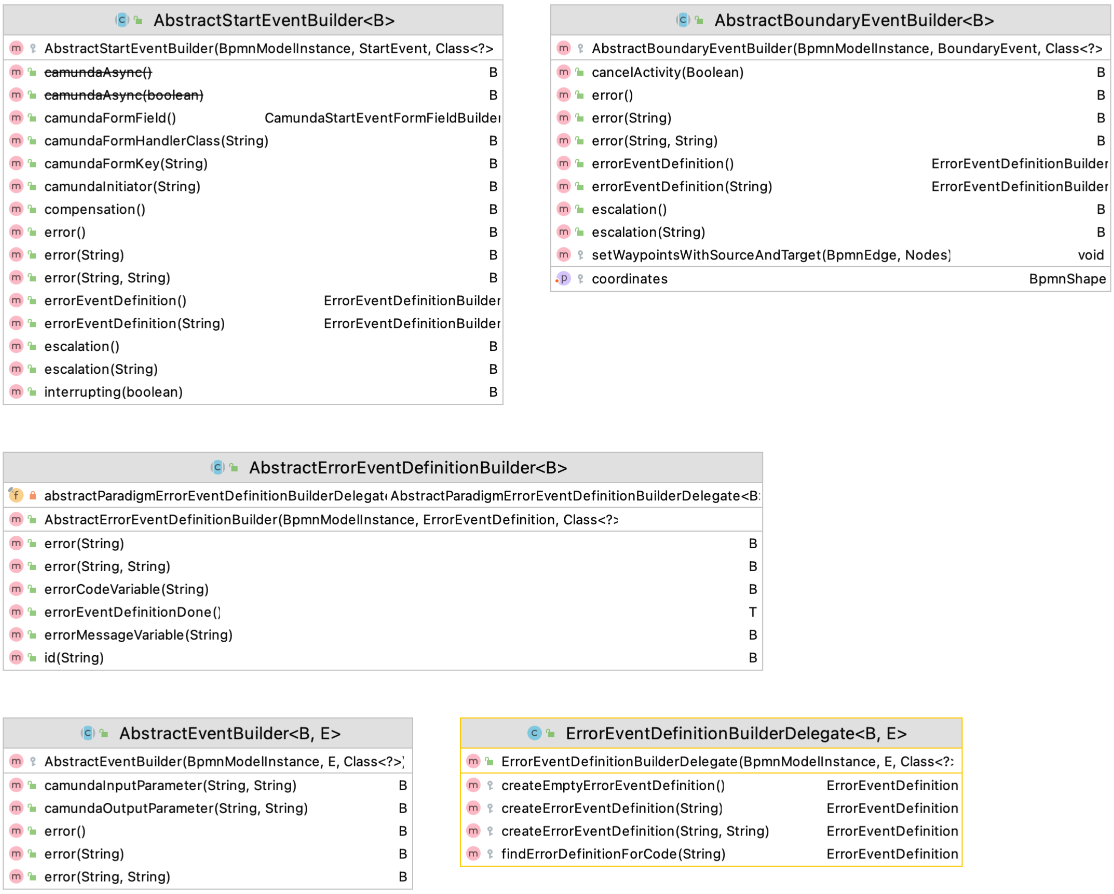


Figure A.4.: Camunda scenario 02 - after refactoring

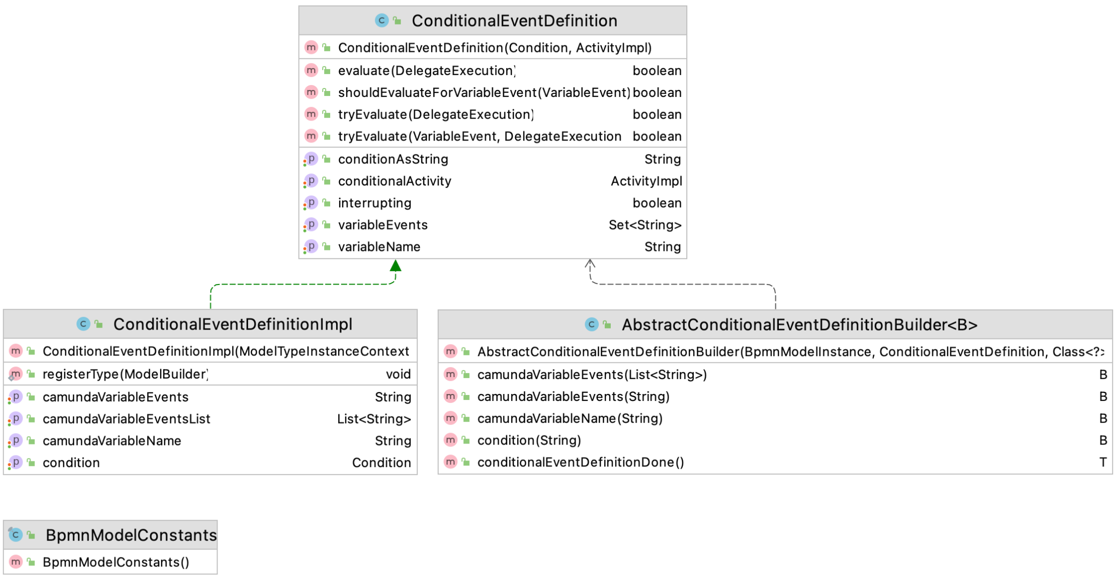


Figure A.5.: Camunda scenario 03 - before refactoring

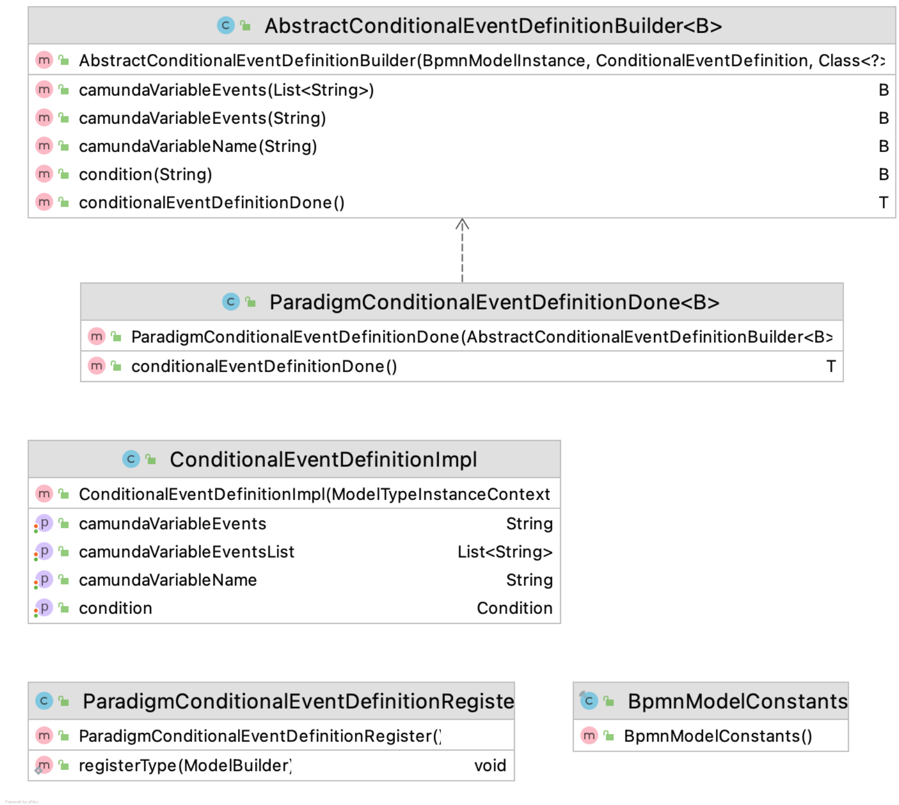


Figure A.6.: Camunda scenario 03 - after refactoring

Figure A.7.: Camunda scenario 04 - before refactoring

Figure A.8.: Camunda scenario 04 - after refactoring

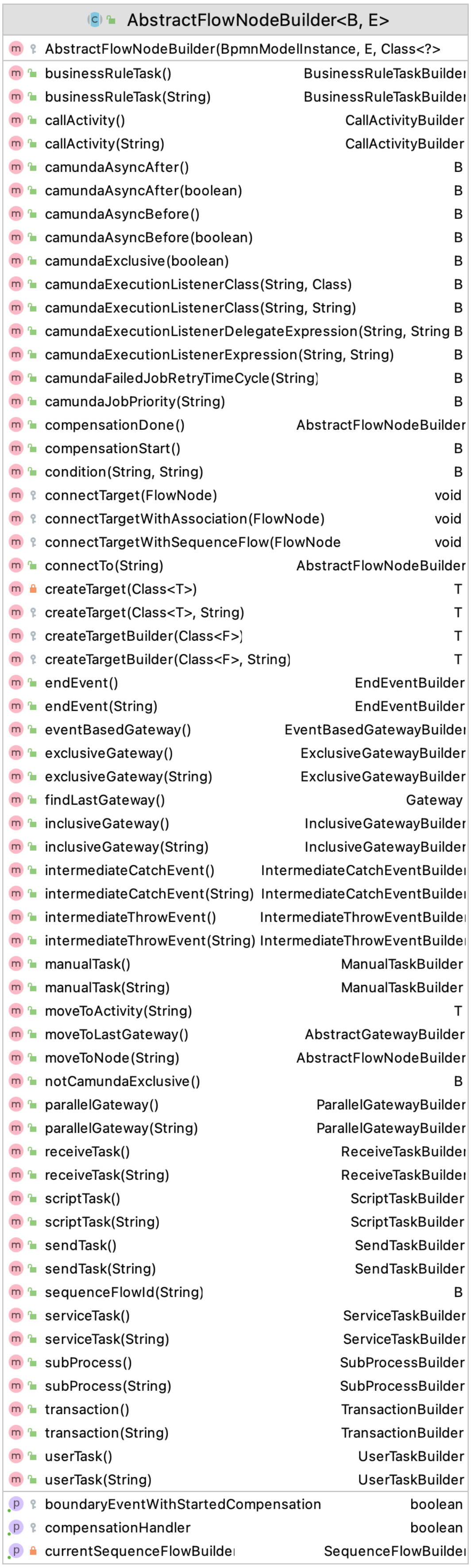


Figure A.9.: Camunda scenario 05 - before refactoring

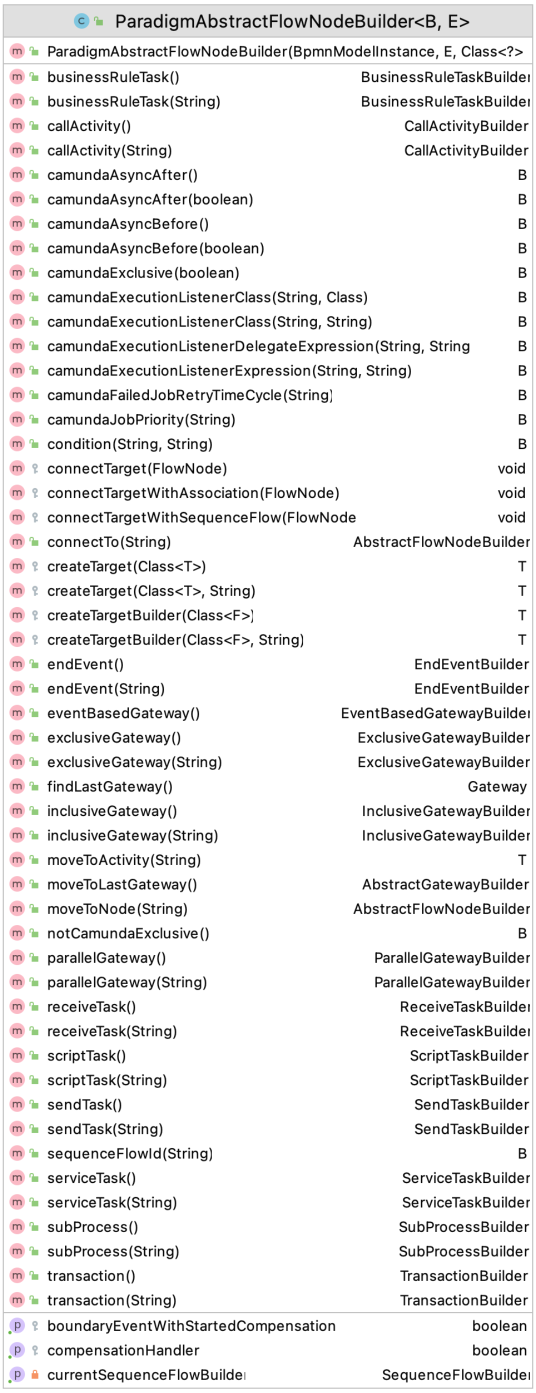


Figure A.10.: Camunda scenario 05 - after refactoring

Figure A.11.: Camunda scenario 06 - before refactoring

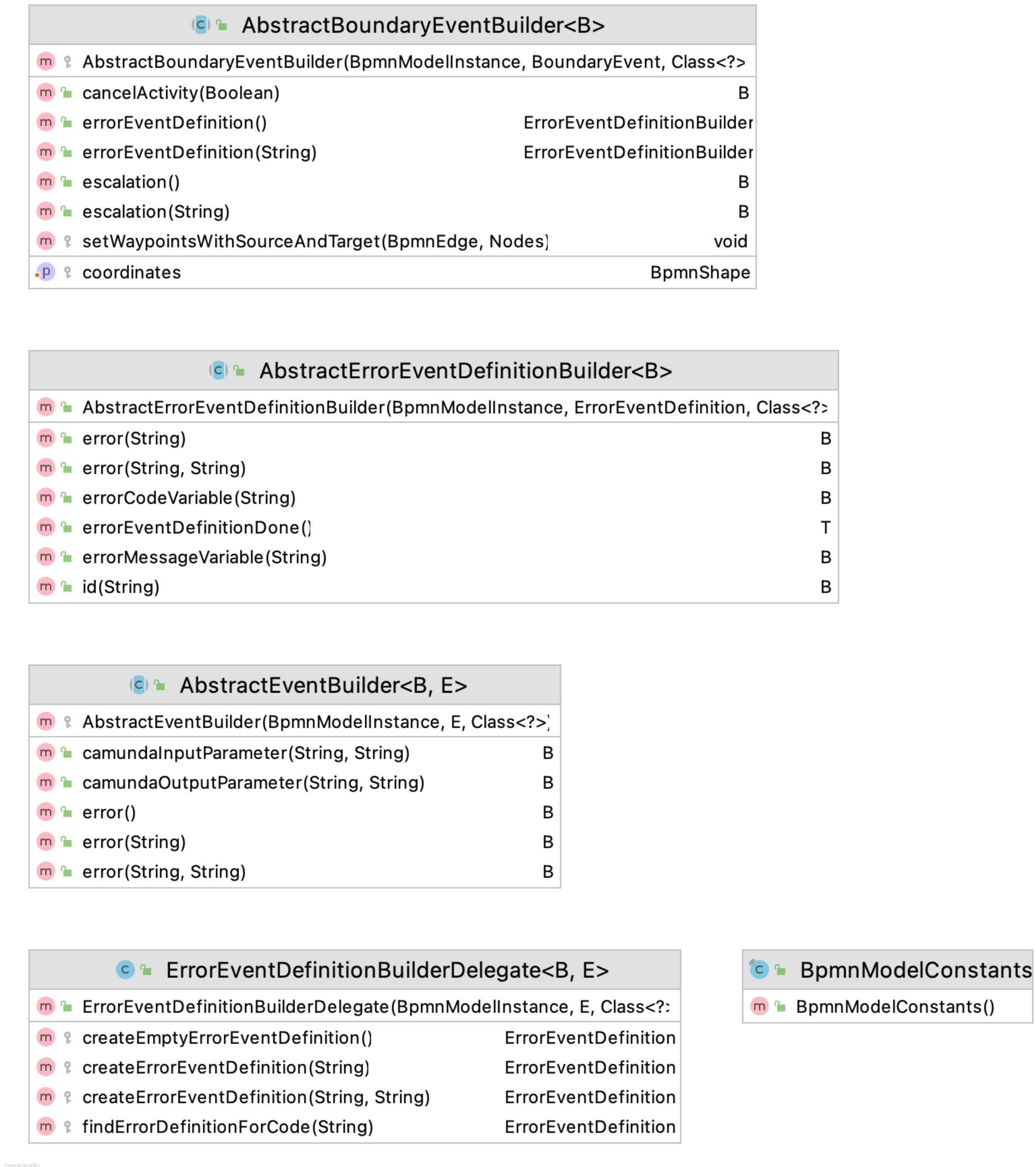


Figure A.12.: Camunda scenario 06 - after refactoring

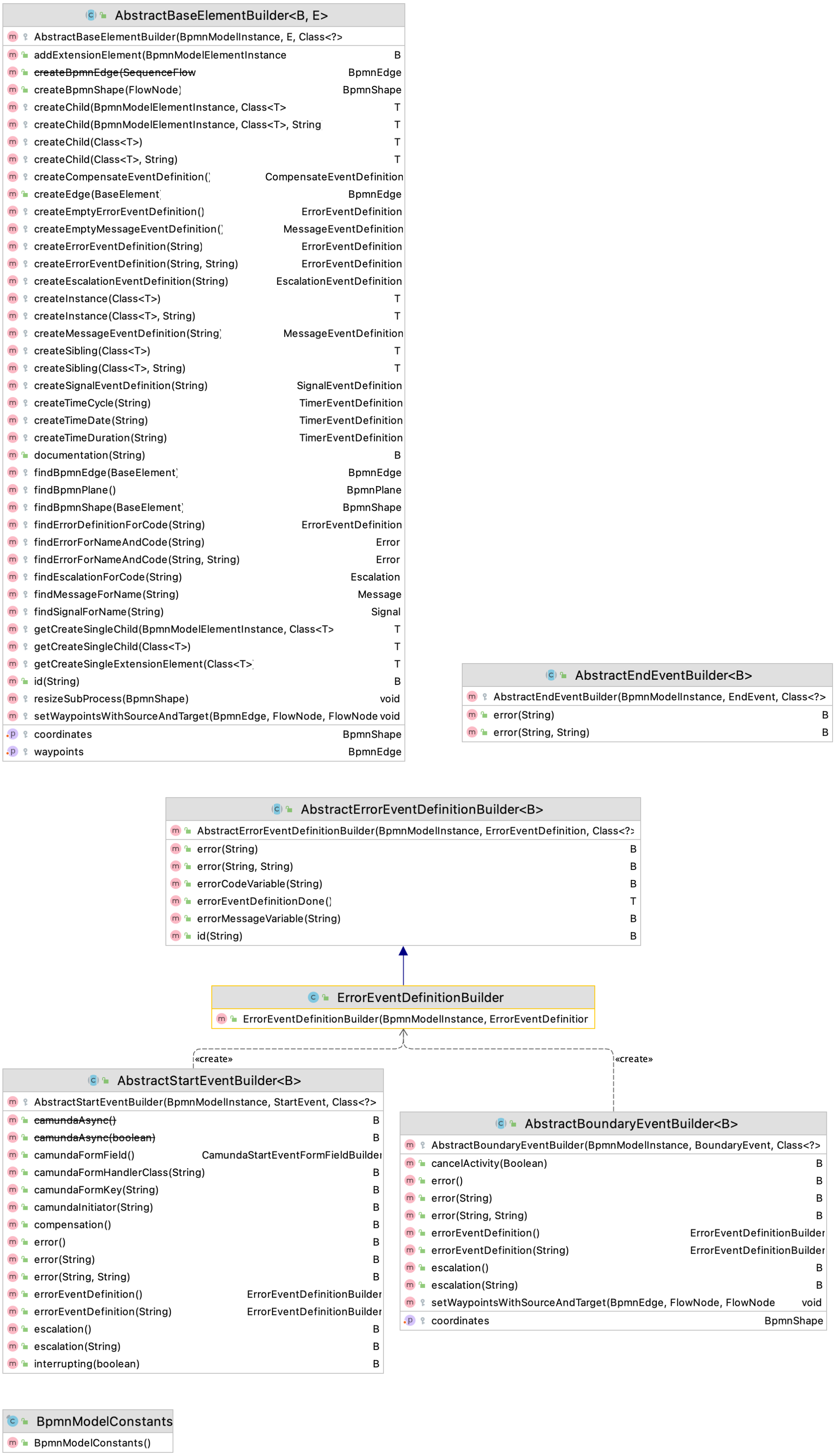


Figure A.13.: Camunda scenario 07 - before refactoring

Figure A.14.: Camunda scenario 07 - after refactoring

Figure A.15.: Camunda scenario 08 - before refactoring

Figure A.16.: Camunda scenario 08 - after refactoring

Figure A.17.: Camunda scenario 09 - before refactoring

Figure A.18.: Camunda scenario 09 - after refactoring

Figure A.19.: Camunda scenario 10 - before refactoring

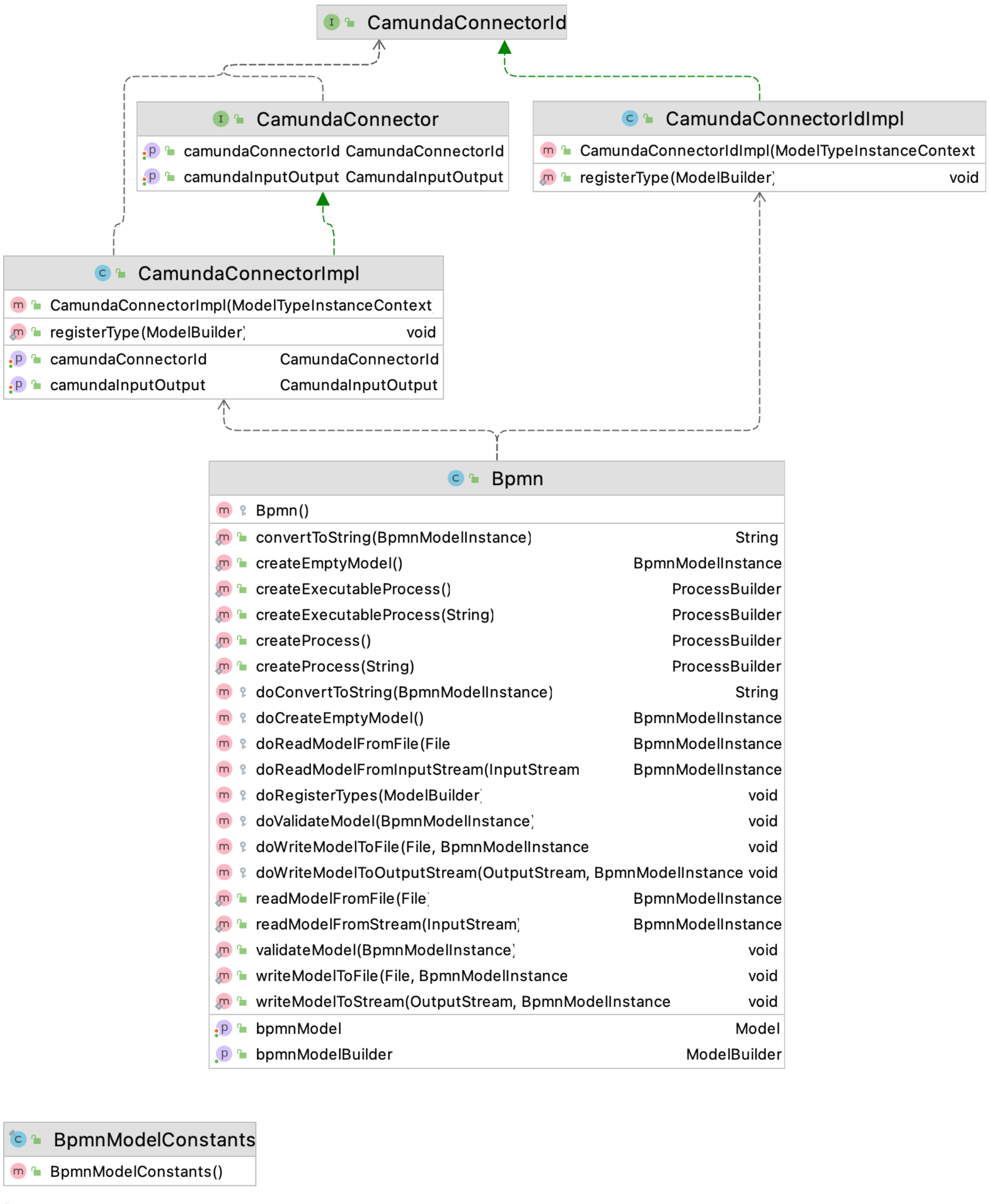


Figure A.20.: Camunda scenario 10 - after refactoring

A.2. KAMP4aPS

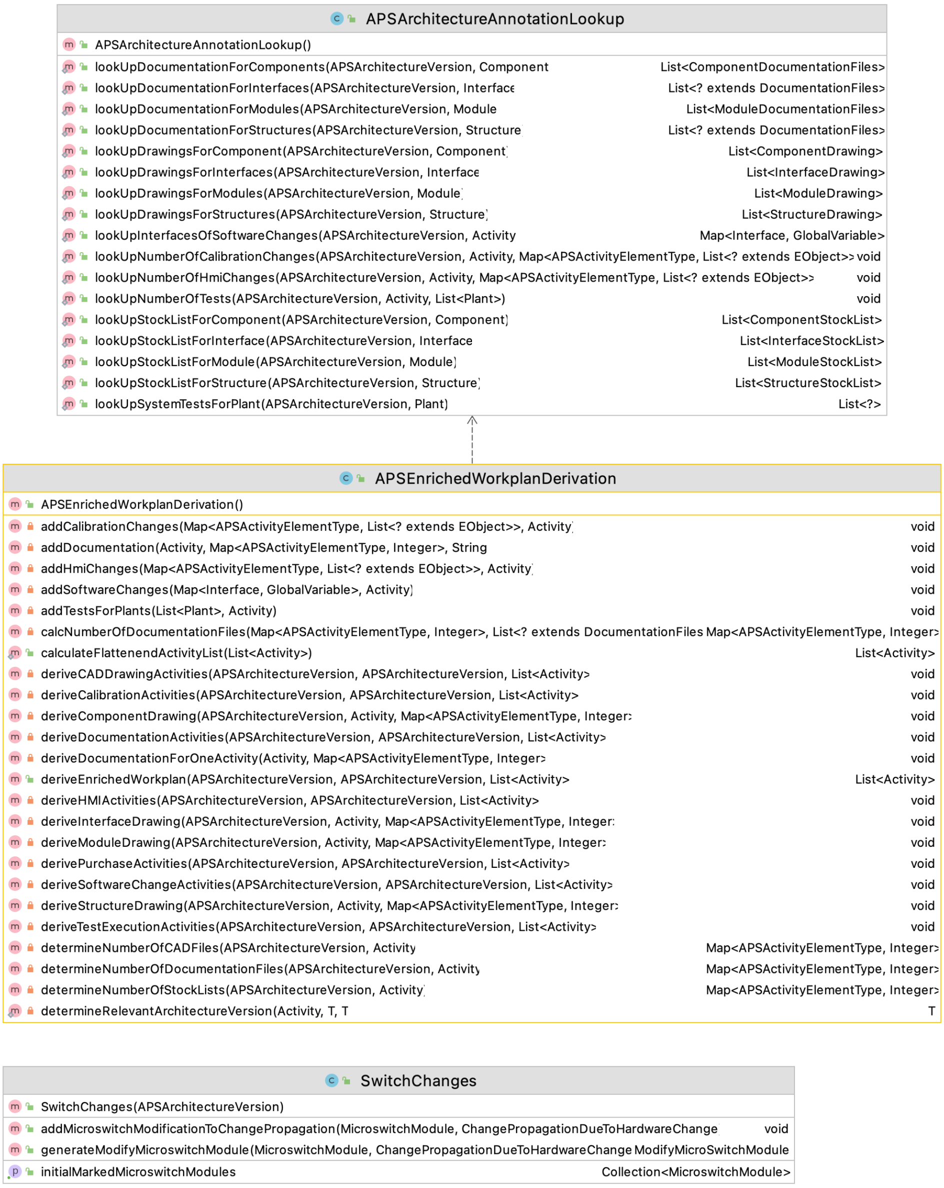


Figure A.21.: KAMP scenario 01 - before refactoring

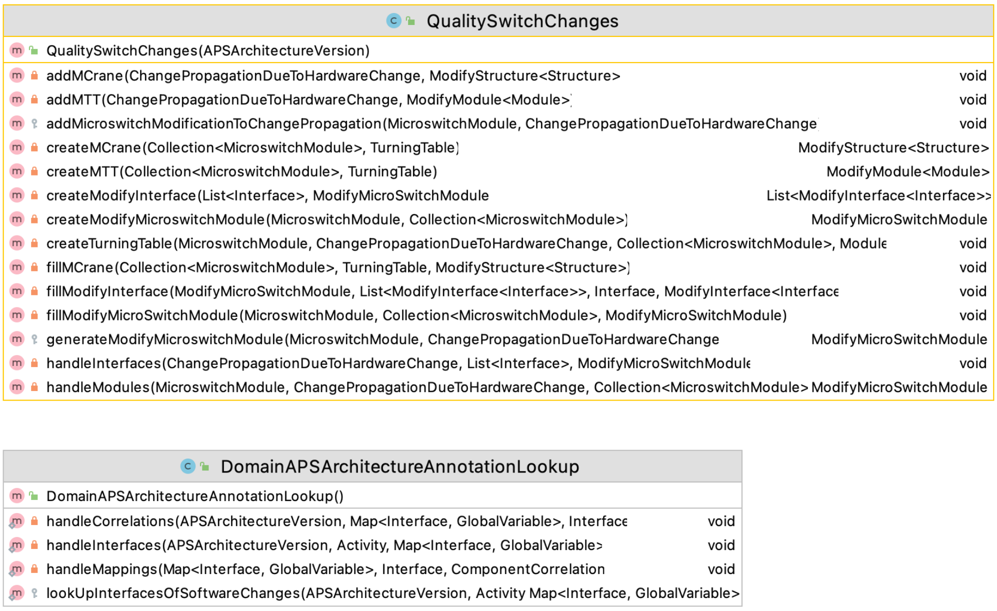


Figure A.22.: KAMP scenario 01 - after refactoring

Figure A.23.: KAMP scenario 02 - before refactoring

Figure A.24.: KAMP scenario 02 - after refactoring

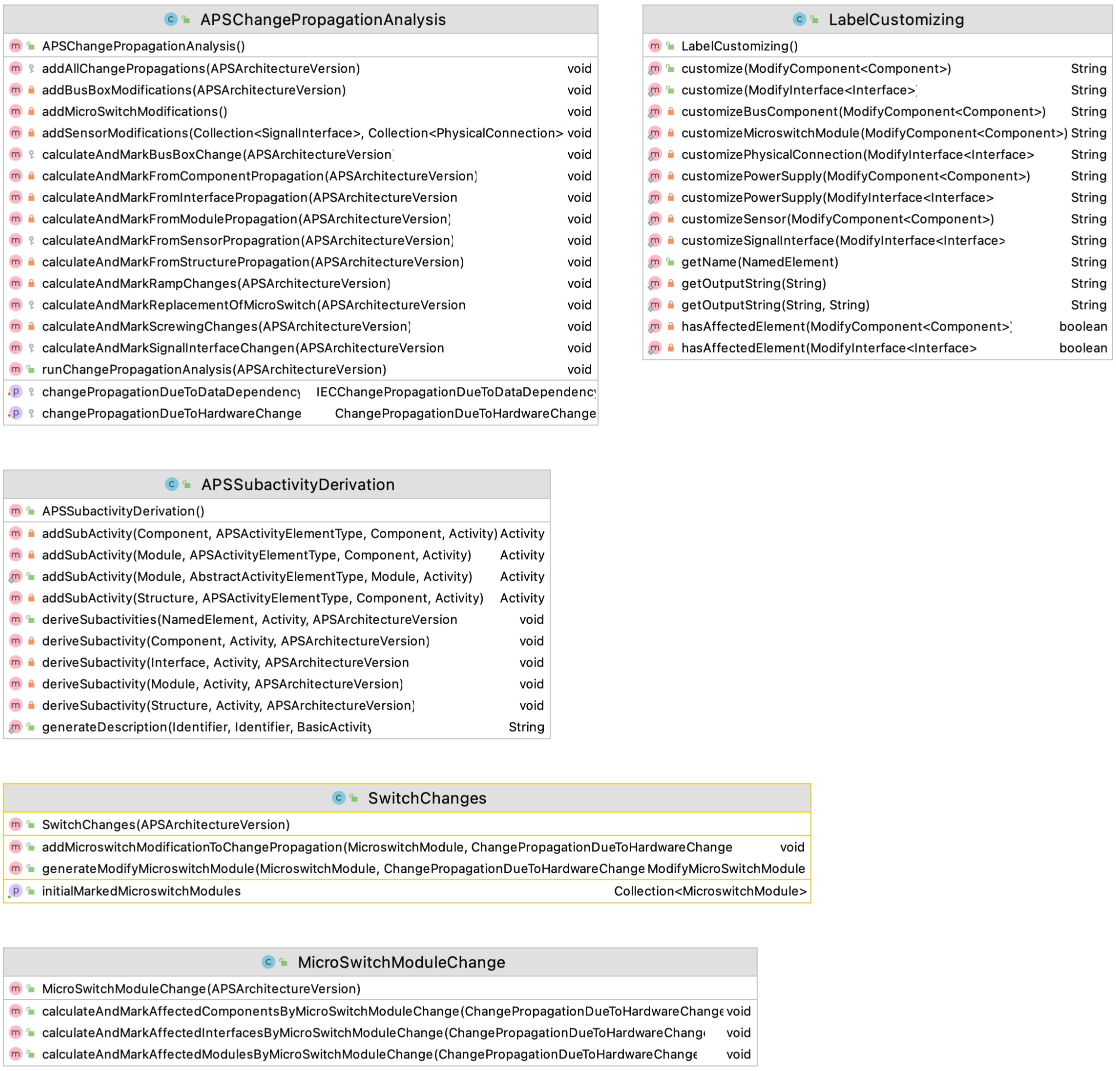


Figure A.25.: KAMP scenario 03 - before refactoring

Figure A.26.: KAMP scenario 03 - after refactoring

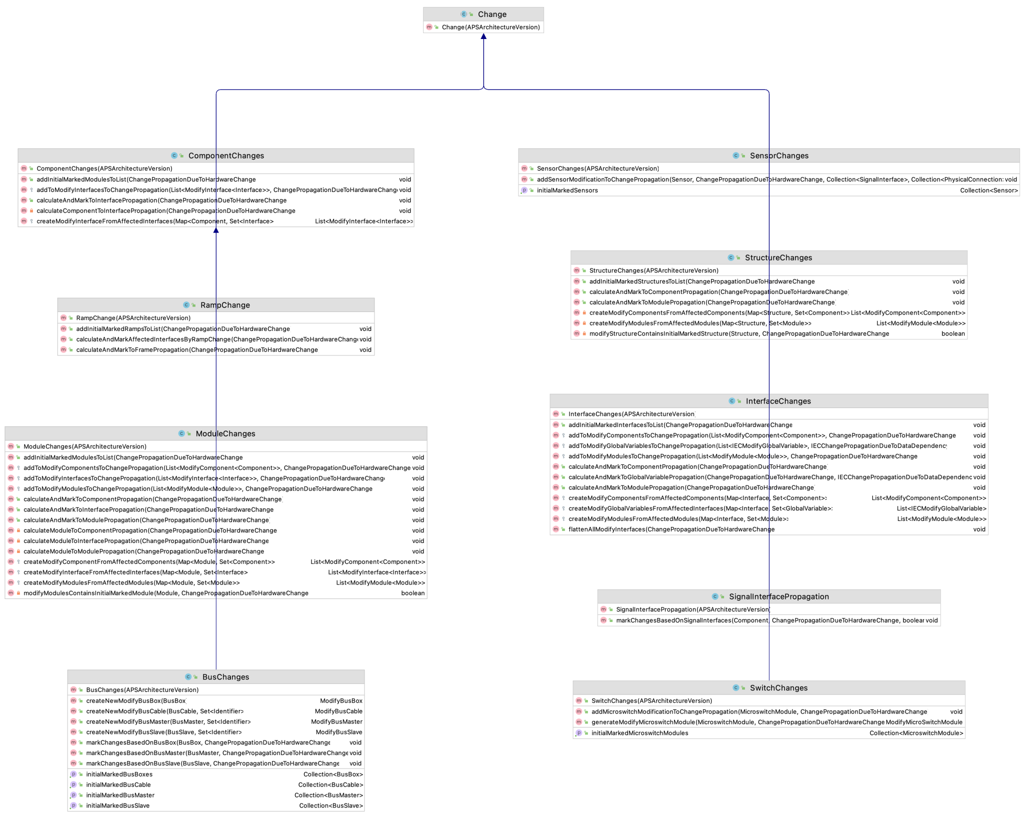


Figure A.27.: KAMP scenario 04 - before refactoring

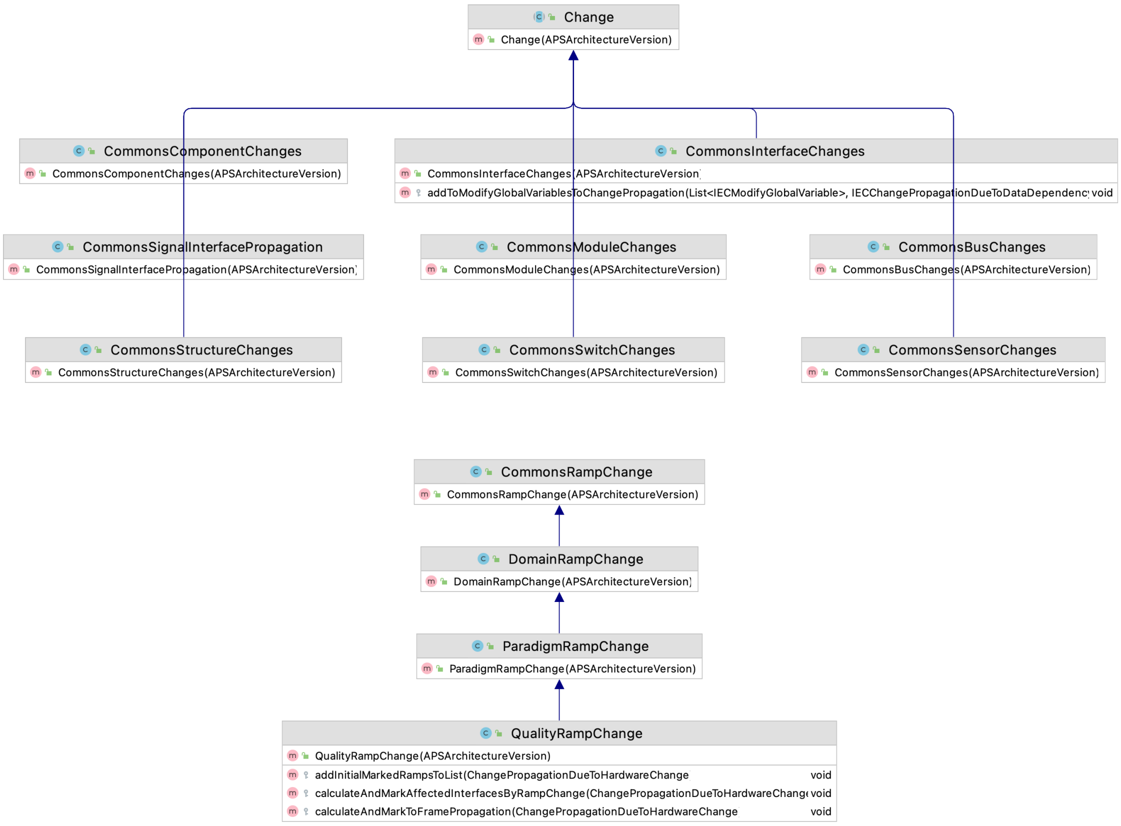


Figure A.28.: KAMP scenario 04 - after refactoring

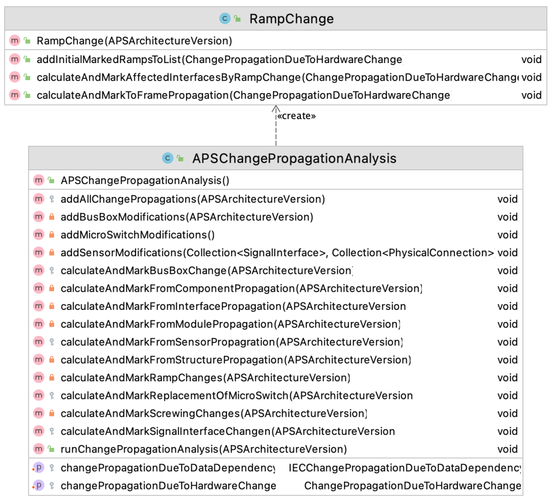


Figure A.29.: KAMP scenario 05 - before refactoring

Figure A.30.: KAMP scenario 05 - after refactoring

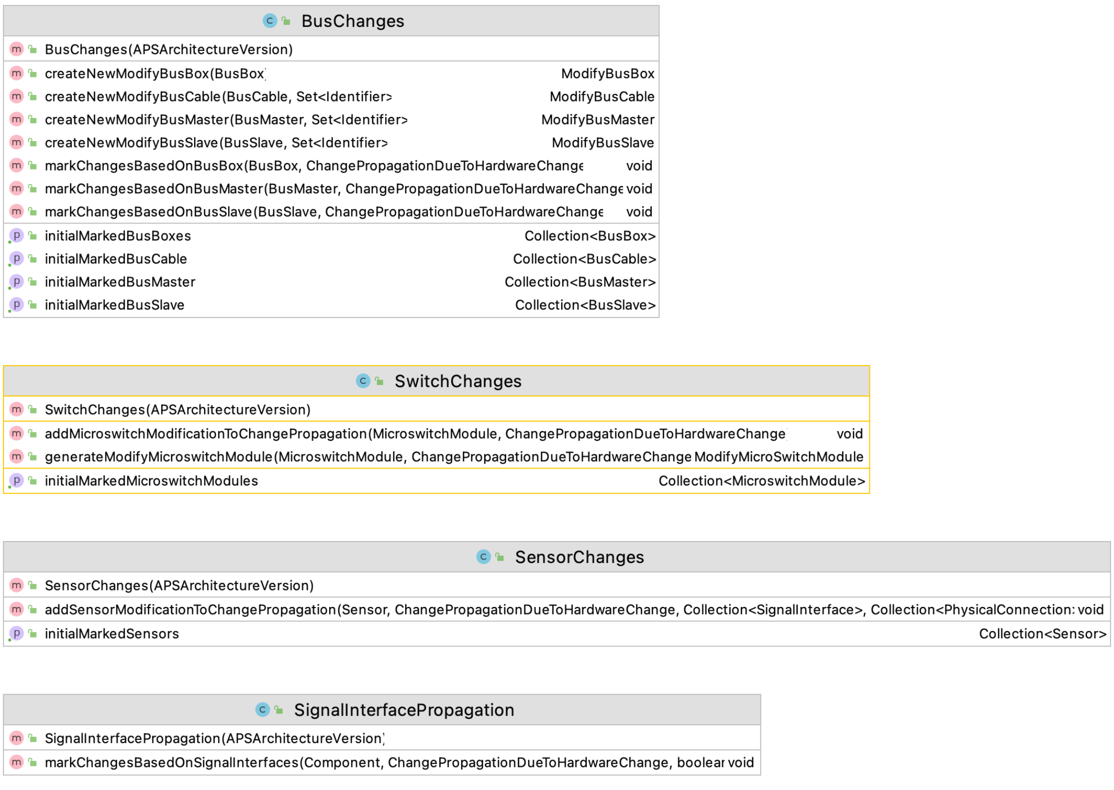


Figure A.31.: KAMP scenario 06 - before refactoring

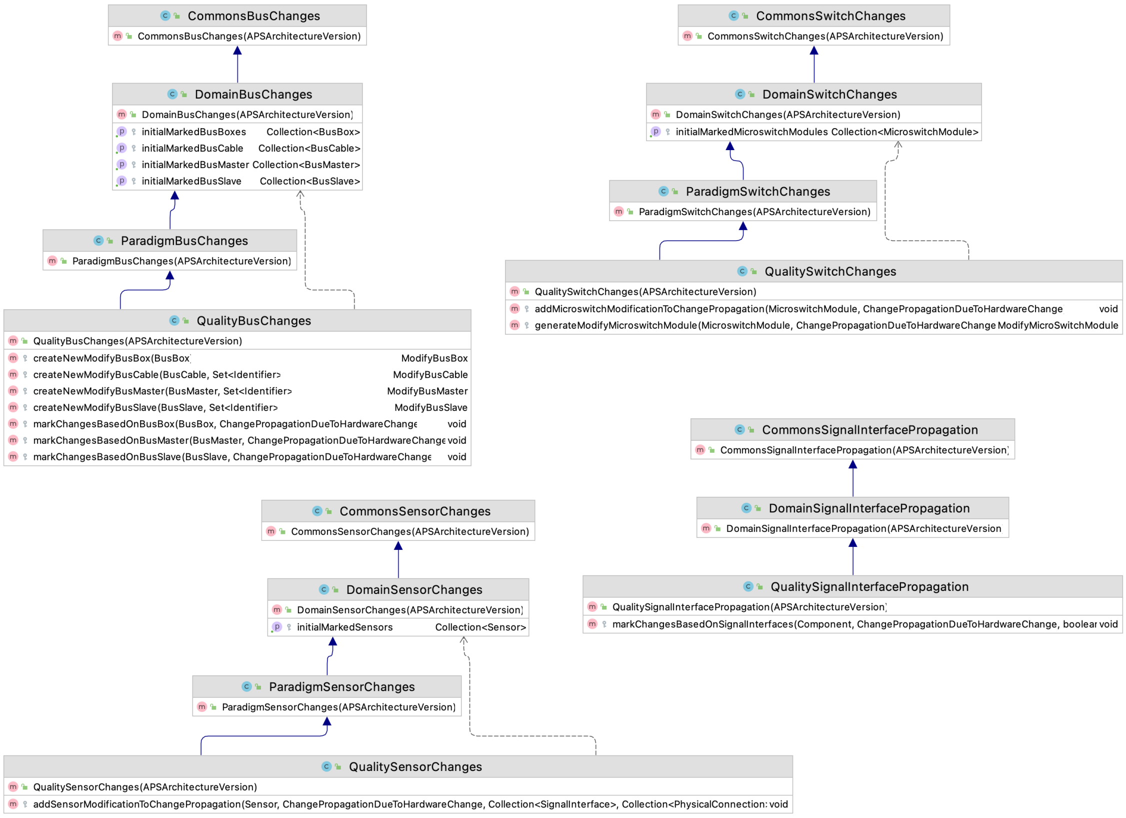


Figure A.32.: KAMP scenario 06 - after refactoring

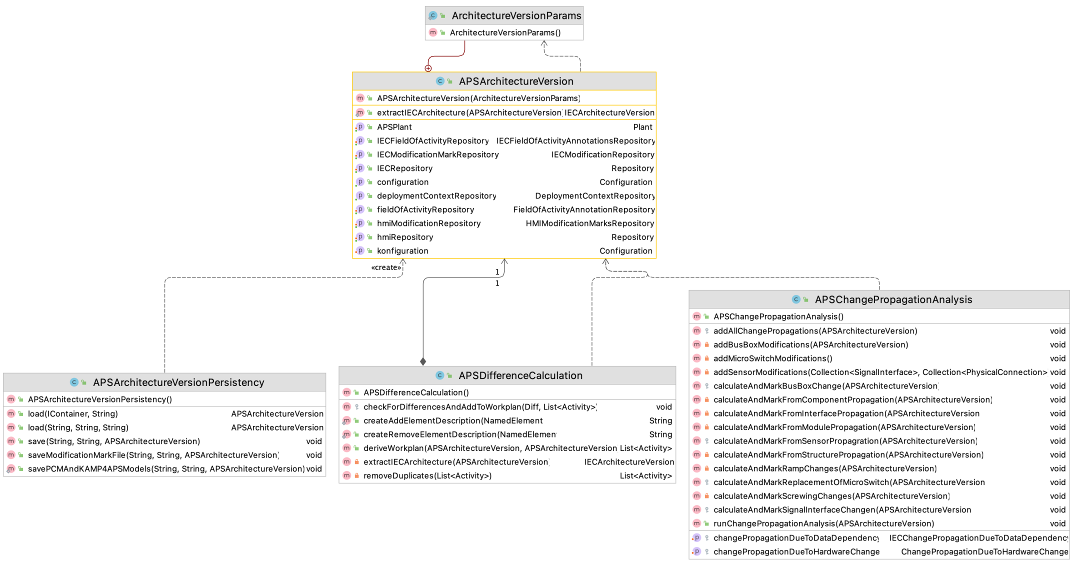


Figure A.33.: KAMP scenario 07 - before refactoring

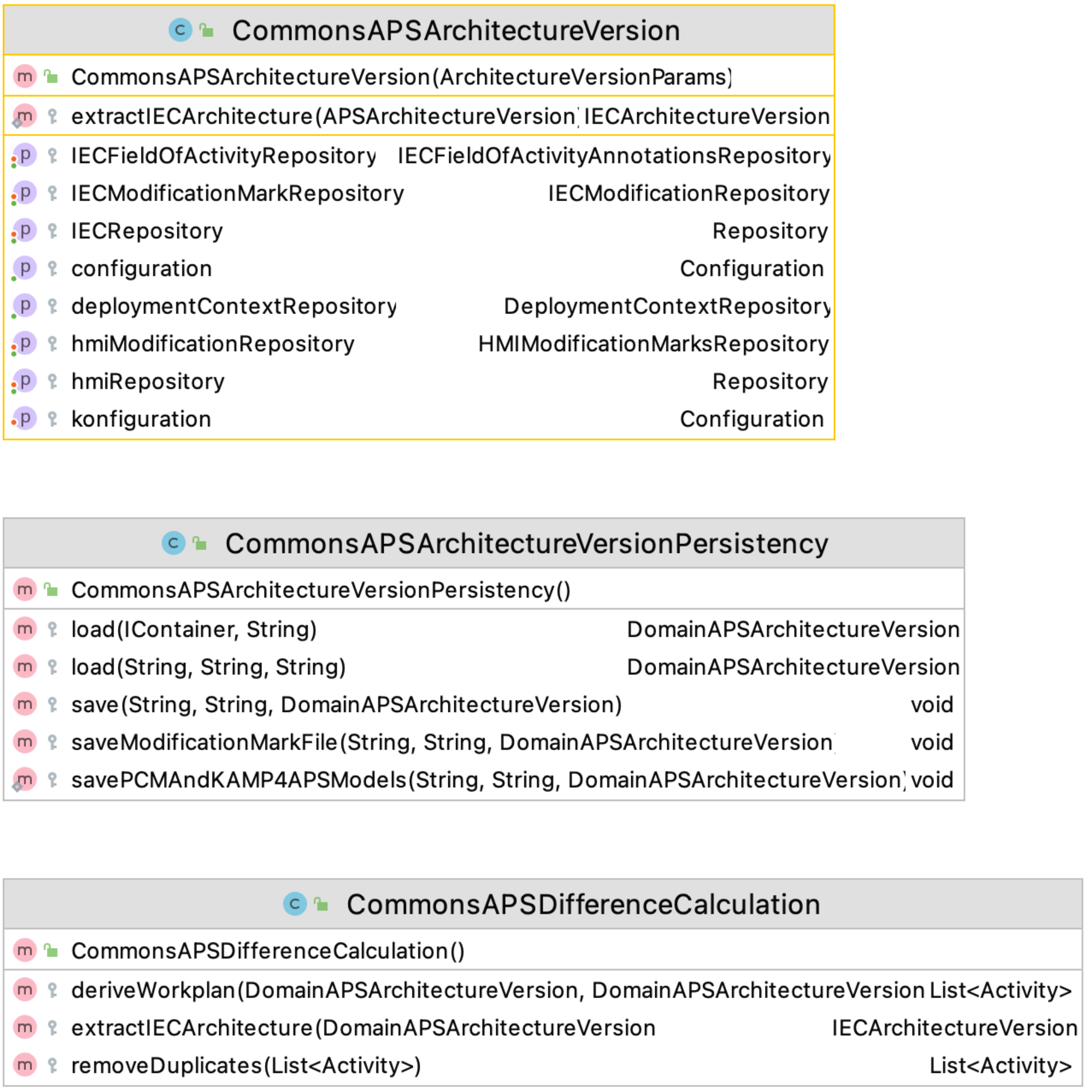


Figure A.34.: KAMP scenario 07 - after refactoring

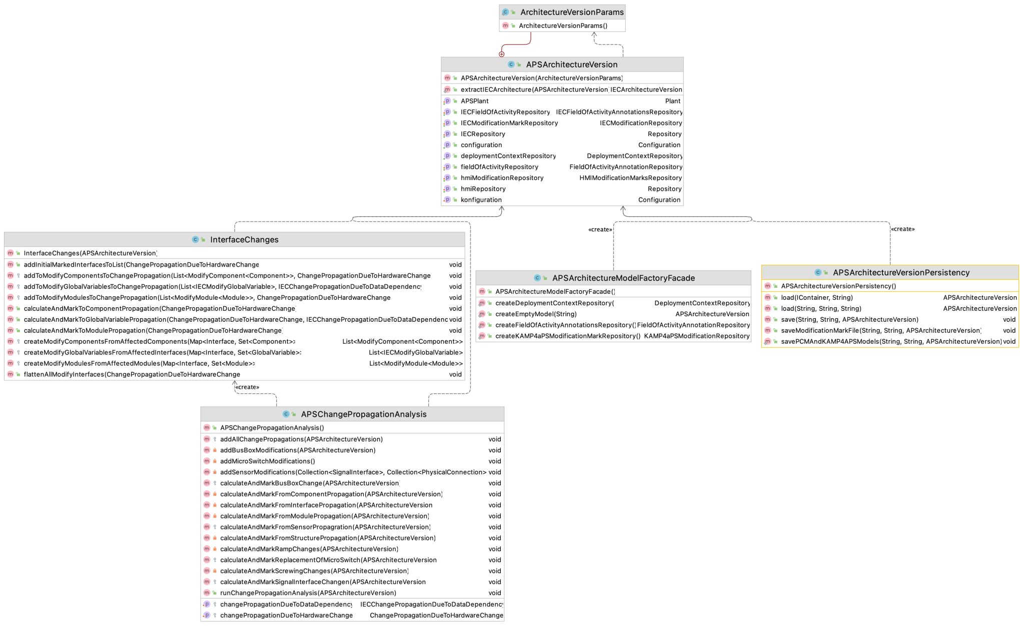


Figure A.35.: KAMP scenario 08 - before refactoring

Figure A.36.: KAMP scenario 08 - after refactoring

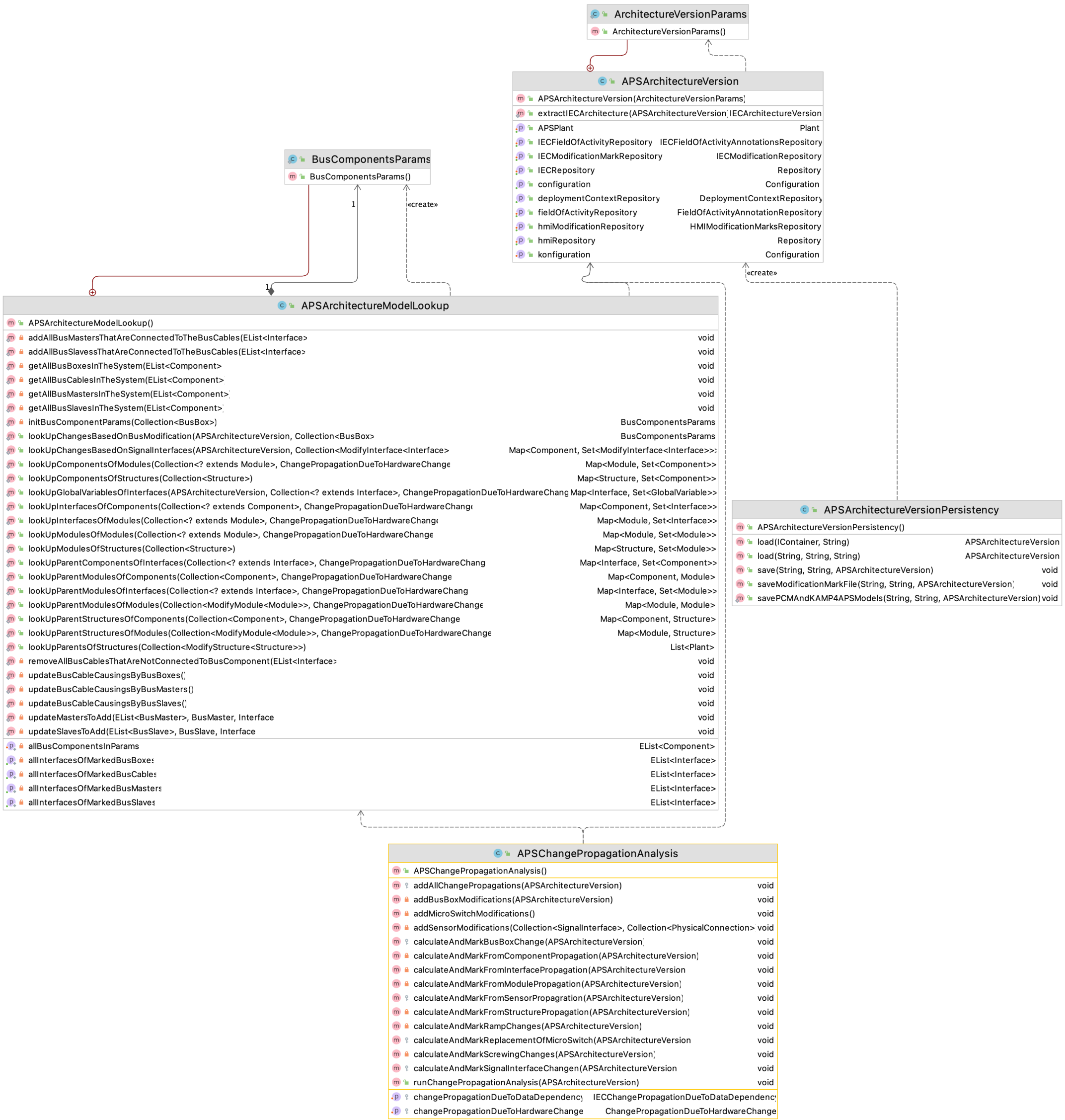


Figure A.37.: KAMP scenario 09 - before refactoring

Figure A.38.: KAMP scenario 09 - after refactoring

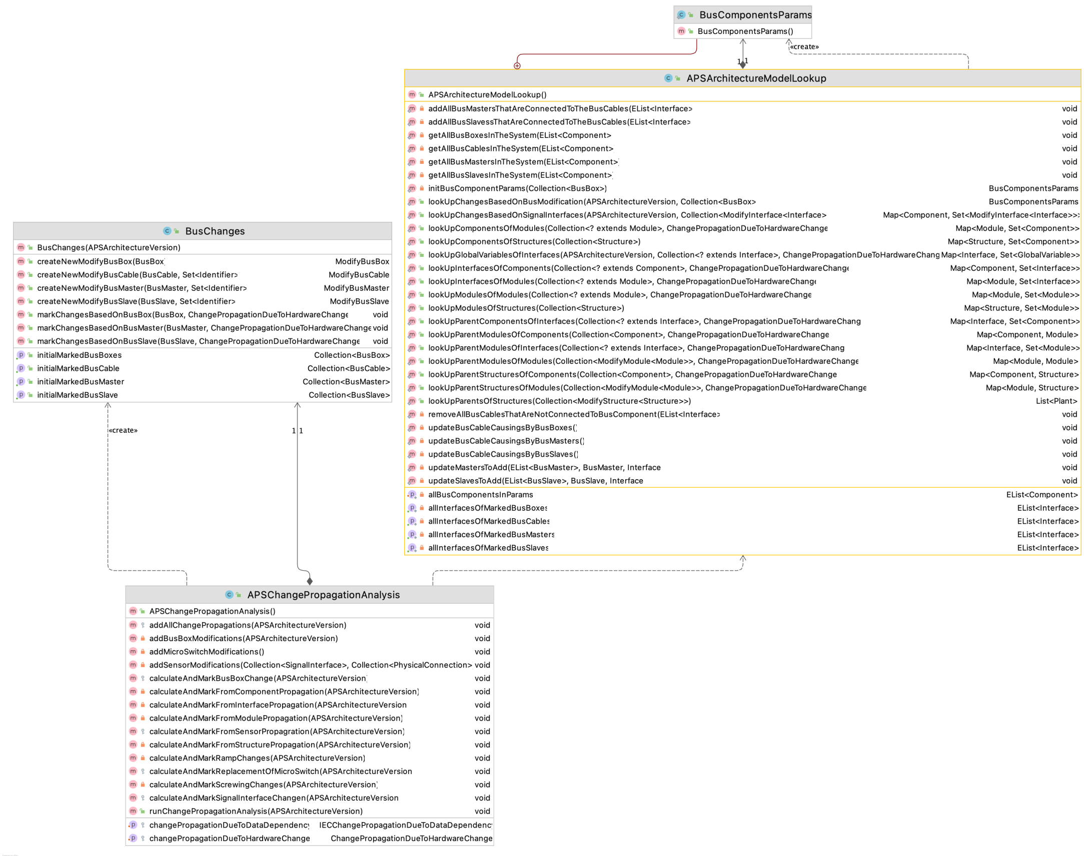


Figure A.39.: KAMP scenario 10 - before refactoring

Figure A.40.: KAMP scenario 10 - after refactoring

A.3. SmartGrid

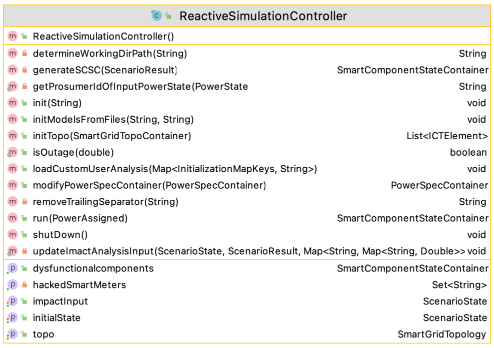


Figure A.41.: SmartGrid scenario 01 - before refactoring

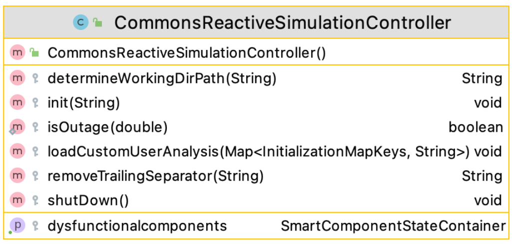


Figure A.42.: SmartGrid scenario 01 - after refactoring

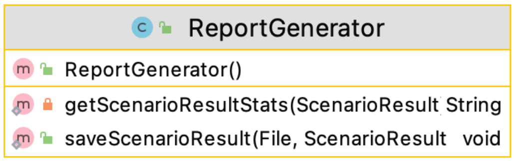


Figure A.43.: SmartGrid scenario 02 - before refactoring

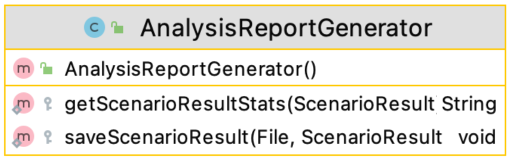


Figure A.44.: SmartGrid scenario 02 - after refactoring

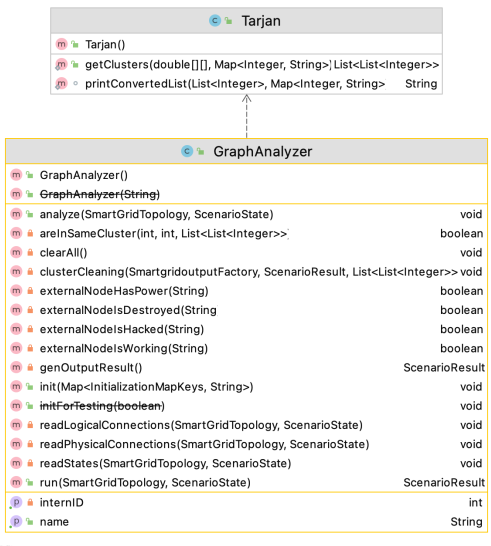


Figure A.45.: SmartGrid scenario 03 - before refactoring

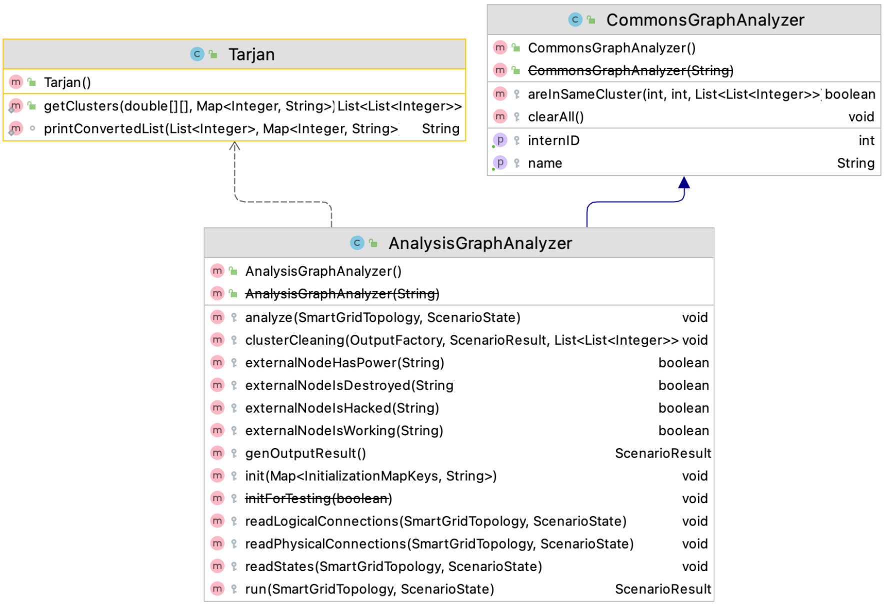


Figure A.46.: SmartGrid scenario 03 - after refactoring

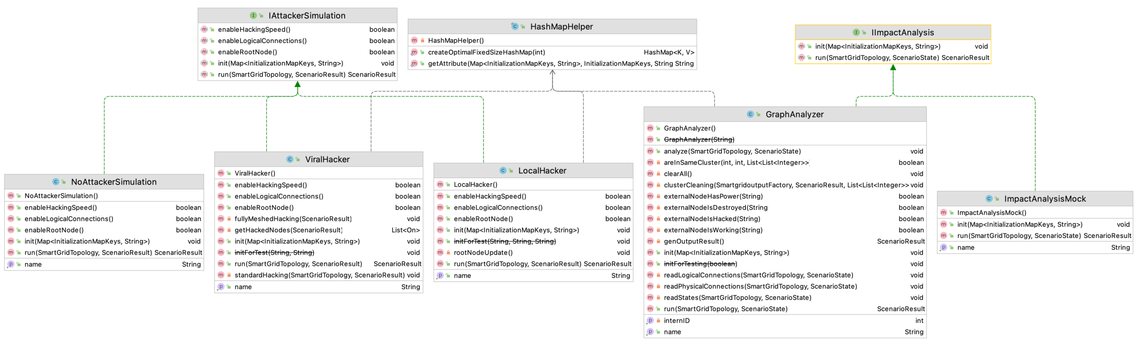


Figure A.47.: SmartGrid scenario 04 - before refactoring

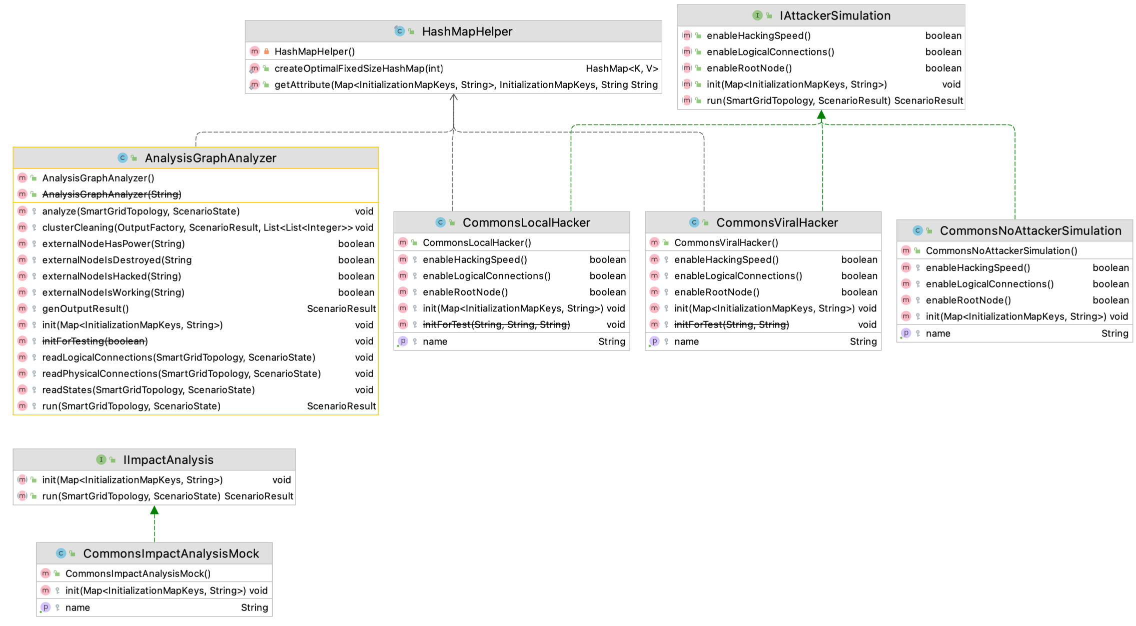


Figure A.48.: SmartGrid scenario 04 - after refactoring

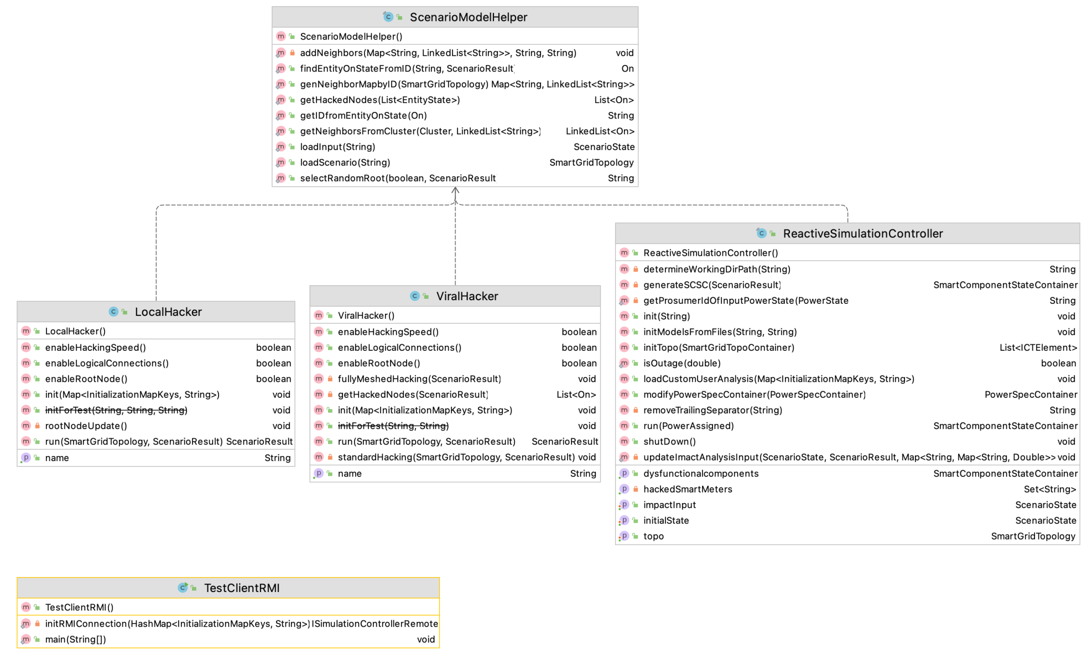


Figure A.49.: SmartGrid scenario 05 - before refactoring

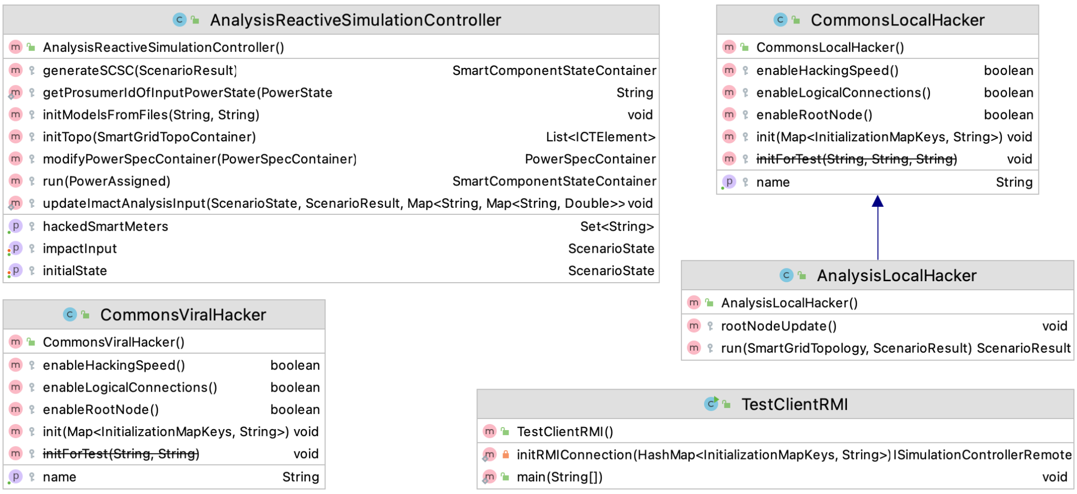


Figure A.50.: SmartGrid scenario 05 - after refactoring

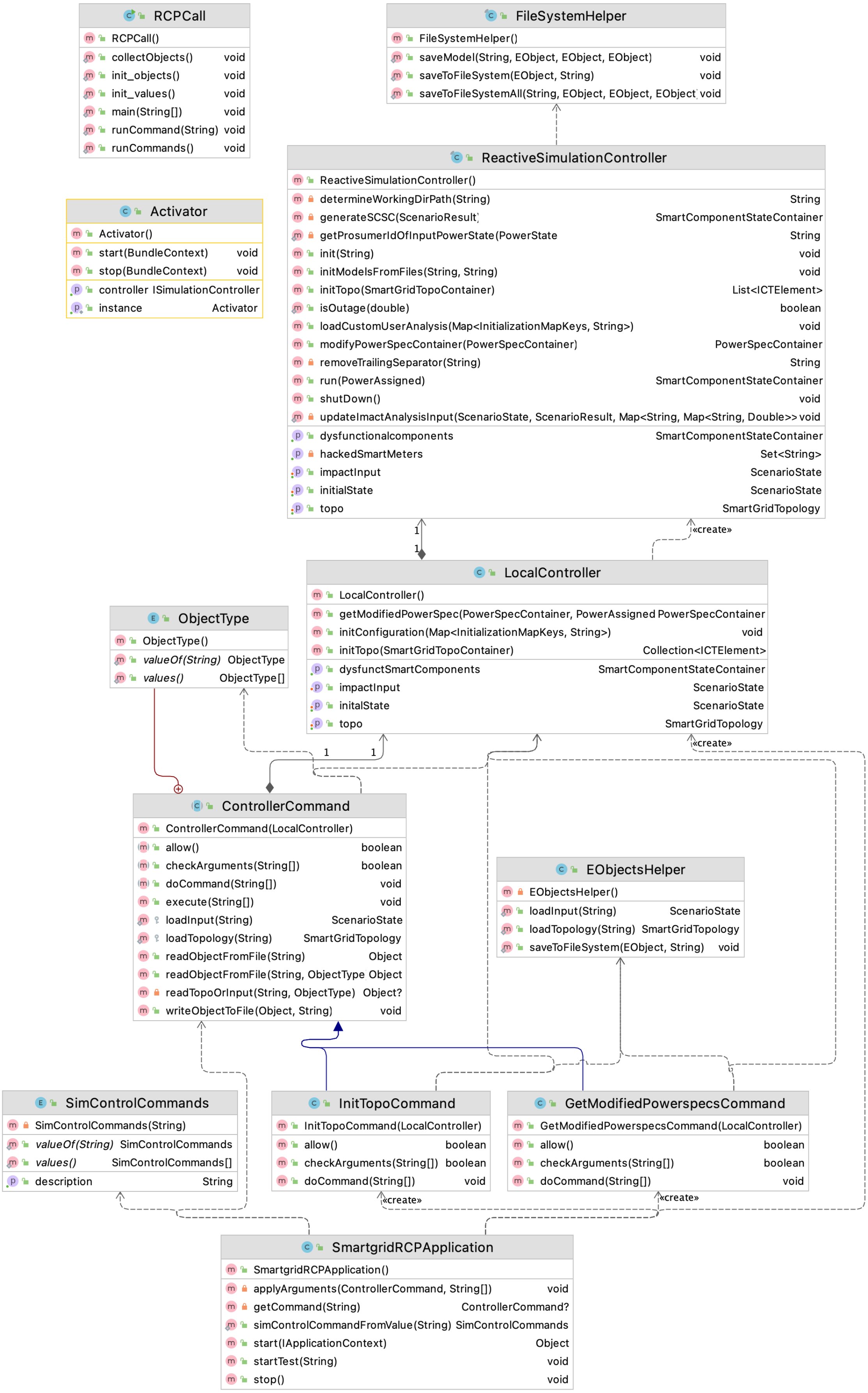


Figure A.51.: SmartGrid scenario 06 - before refactoring

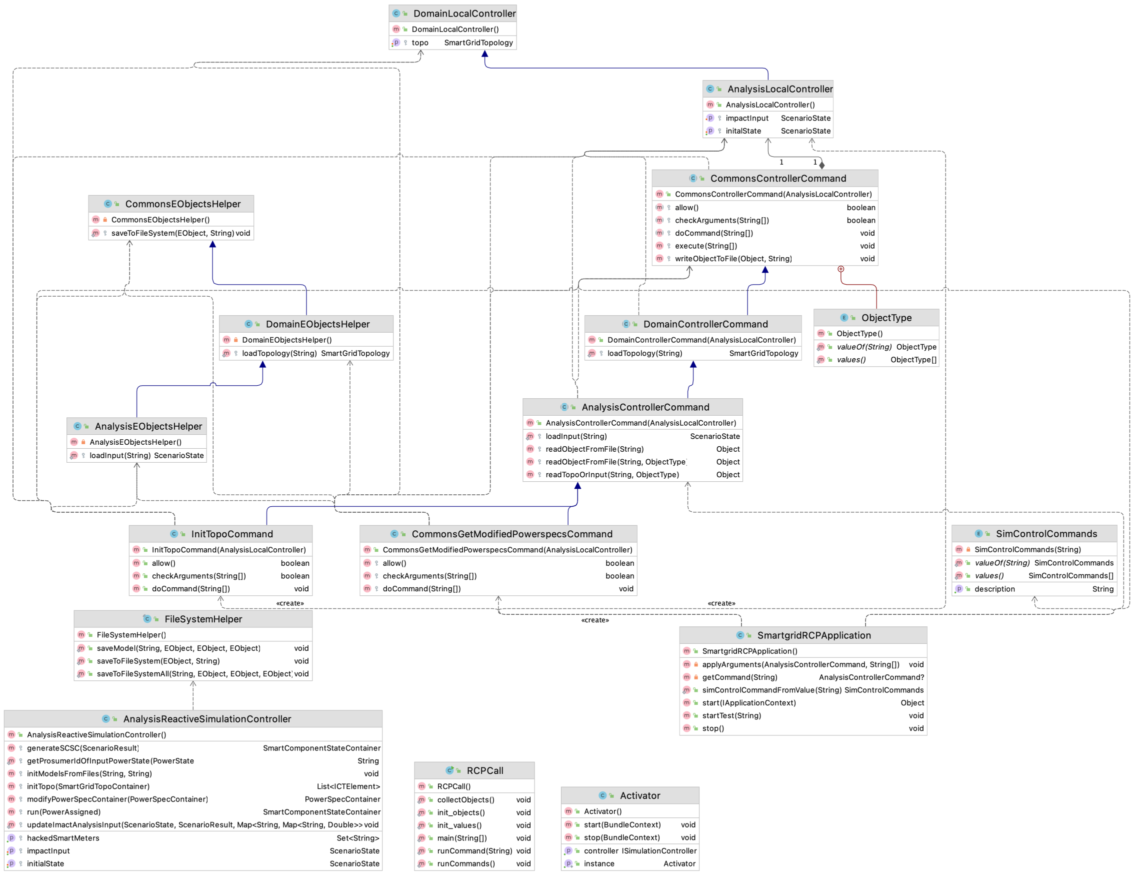


Figure A.52.: SmartGrid scenario 06 - after refactoring

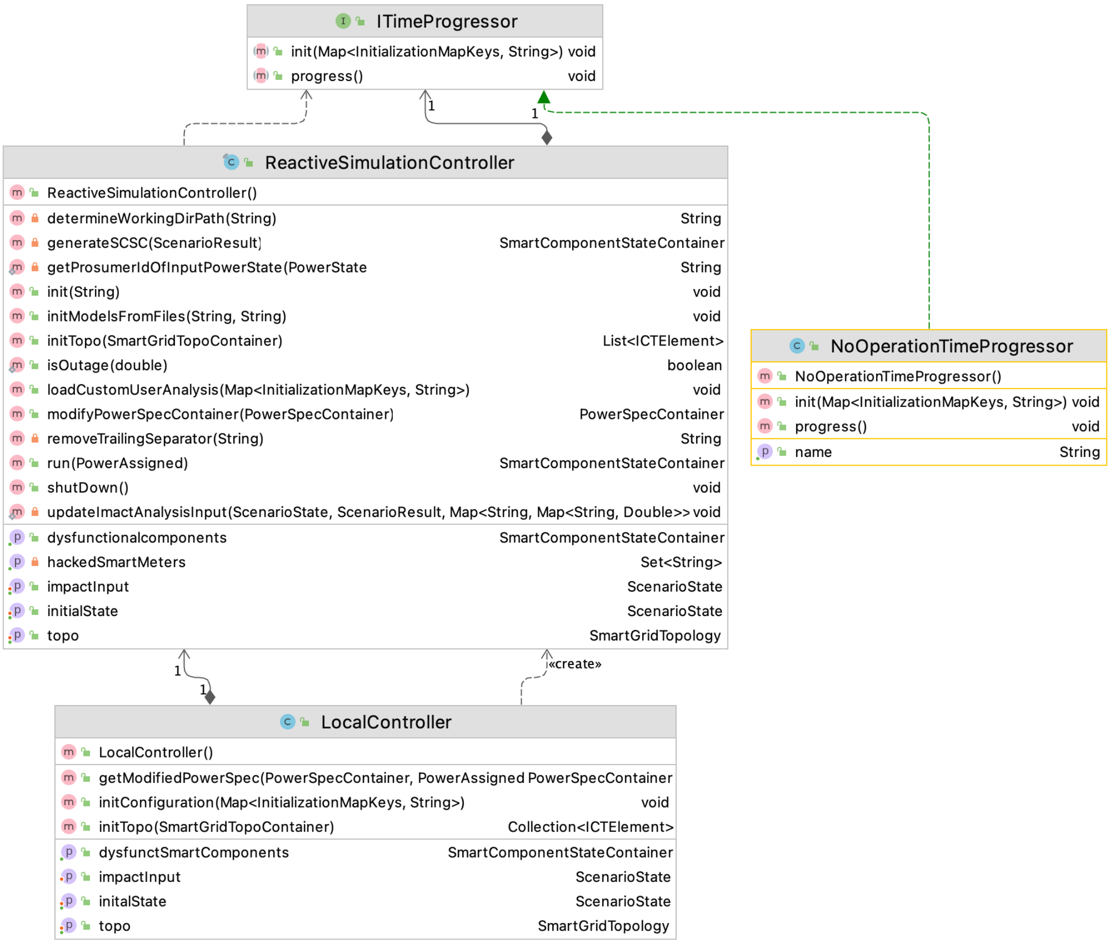


Figure A.53.: SmartGrid scenario 07 - before refactoring

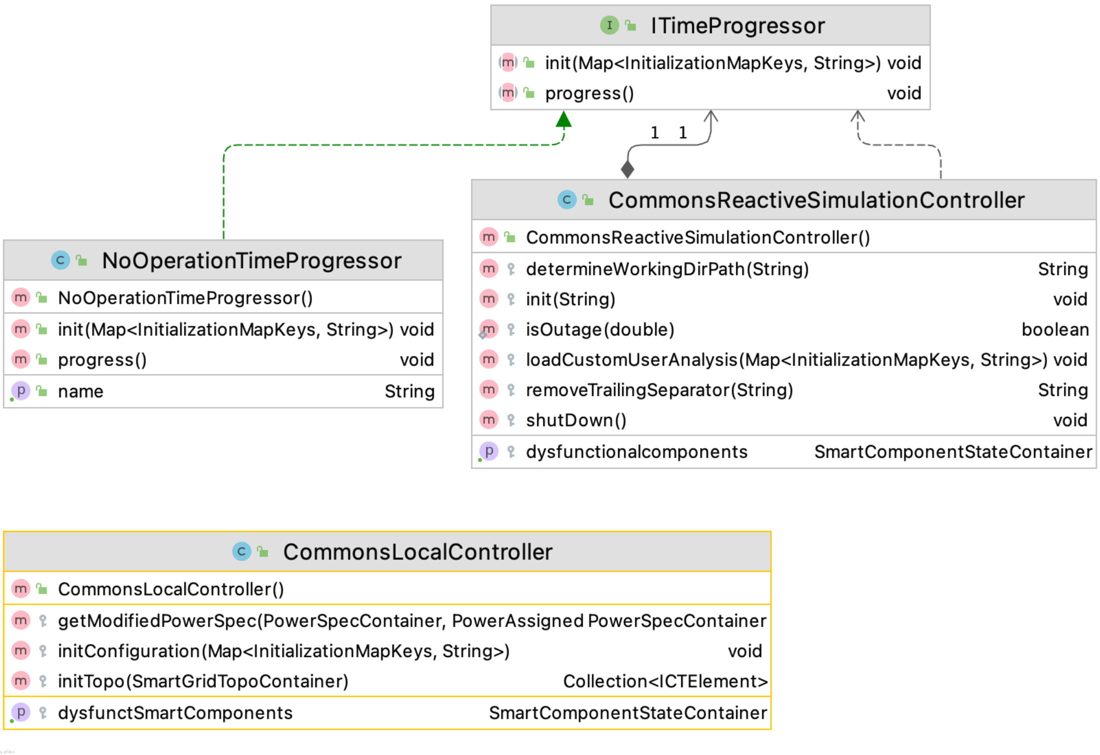


Figure A.54.: SmartGrid scenario 07 - after refactoring

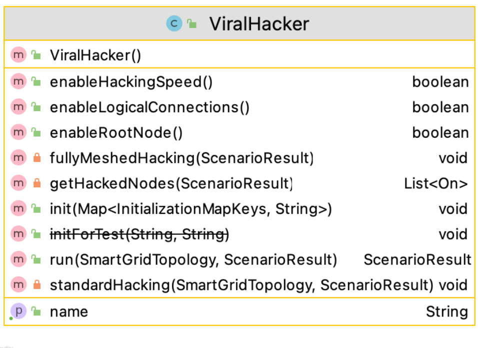


Figure A.55.: SmartGrid scenario 08 - before refactoring

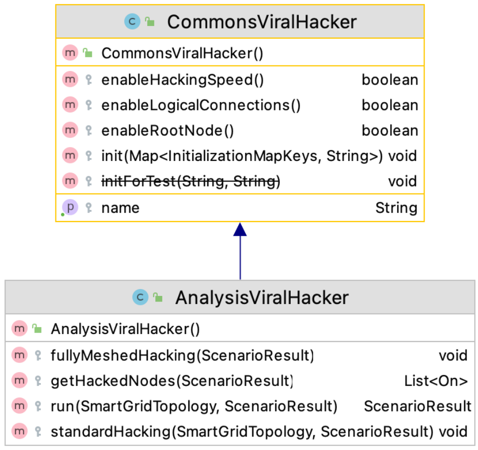


Figure A.56.: SmartGrid scenario 08 - after refactoring

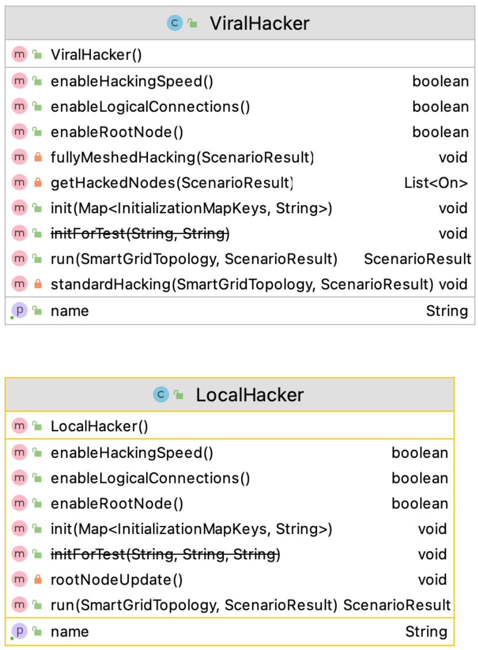


Figure A.57.: SmartGrid scenario 09 - before refactoring

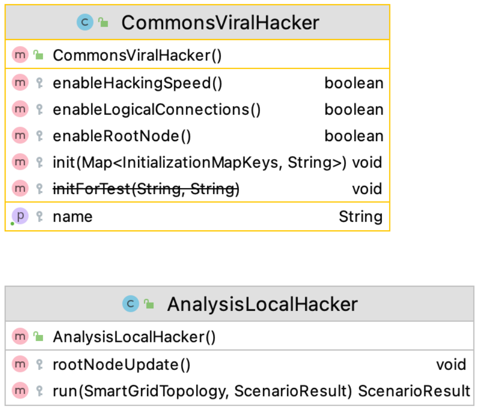


Figure A.58.: SmartGrid scenario 09 - after refactoring

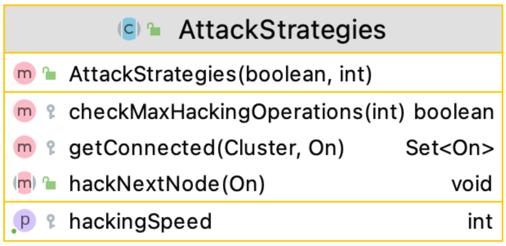


Figure A.59.: SmartGrid scenario 10 - before refactoring

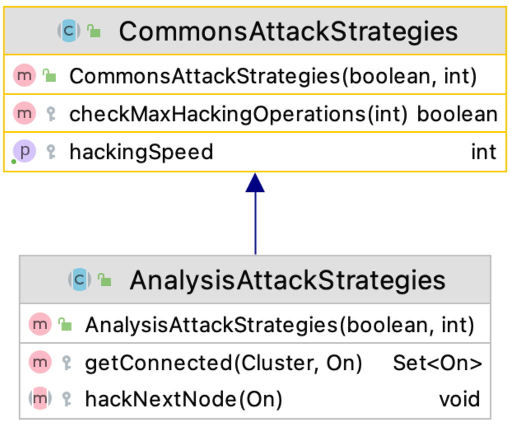


Figure A.60.: SmartGrid scenario 10 - after refactoring

KIT Scientific Working Papers
ISSN 2194-1629