



UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Faculty of Computer Science, Electrical Engineering and Mathematics
Heinz Nixdorf Institute and Department of Computer Science
Software Engineering Research Group
Zukunftsmeile 1
33102 Paderborn

Extensible Performance Prototype Transformations for Multiple Platforms

Bachelor's Thesis

Submitted to the Software Engineering Research Group
in Partial Fulfillment of the Requirements for the
Degree of
Bachelor of Science

by
CHRISTIAN KLAUSSNER
Andreasstr. 15
33098 Paderborn

Thesis Advisor:
Sebastian Lehrig

Thesis Supervisors:
Jun.-Prof. Dr.-Ing. Steffen Becker
and
Prof. Dr. Gerd Szwillus

Paderborn, July 2014

Abstract

Performance engineers analyze the performance of software architectures before their actual implementation to resolve performance bottlenecks in early development phases. Performance prototyping is such an approach where software architecture models are transformed to runnable performance prototypes that can provide analysis data for a specific target operation platform. This coupling to the operation platform comprises new challenges for performance prototyping in the context of cloud computing because a variety of different cloud platforms exists. Because the choice of platform impacts performance, this variety of platforms induces the need for prototype transformations that either support several platforms directly or that are easily extensible. However, current performance prototyping approaches are tied to only a small set of concrete platforms and lack an investigation of their extensibility for new platforms, thus rendering performance prototyping ineffective for cloud computing.

To cope with this problem, this thesis extends the performance prototyping approach ProtoCom by an additional target platform, namely the SAP HANA Cloud, and analyzes its extensibility during the extension process. This analysis reveals that the extensibility of ProtoCom can be improved in several areas. Furthermore, it points to opportunities for future enhancements and extensions, based on systematically gathered, quantitative data.

Declaration

(Translation from German)

I hereby declare that I prepared this thesis entirely on my own and have not used outside sources without declaration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This thesis has not been submitted in the same or substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

Original Declaration Text in German:

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

City, Date

Signature

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Solution	3
2	Fundamentals	5
2.1	Performance Prototyping	5
2.2	Xtend 2	6
2.2.1	Syntax and Semantics	7
2.2.2	Templates	8
2.3	Architecture-Centric MDSD	10
2.4	Goal Question Metric (GQM) Method	11
2.5	Architecture-Level Modifiability Analysis	12
3	Definition Phase	15
3.1	Goal Definition	15
3.2	ProtoCom Architecture	15
3.3	Extension Scenarios	16
3.3.1	High-Level Extension Scenarios	18
3.3.2	Transformation Extensions	19
3.3.3	Framework Extensions	20
3.4	Questions, Hypotheses, and Metrics	21
3.4.1	Transformations	21
3.4.2	Framework	22
3.4.3	Metrics	22
4	Collection Phase	25
4.1	Mapping the PCM to Java	25
4.1.1	Language Constructs	25
4.1.2	Proxy and Context Patterns	26
4.2	Reference Implementation	27
4.2.1	PCM Reference Instance	27
4.2.2	Transformed Model	29
4.2.3	HTTP Registry	31
4.2.4	Communication Protocol	32
4.2.5	Web User Interface	33
4.2.6	Usage Scenarios	34

4.3	Transformations	35
4.3.1	Code Generation	36
4.3.2	Entity Transformations	39
5	Interpretation Phase	43
5.1	Answers to Questions	43
5.1.1	Language Component	43
5.1.2	Traversal Framework	46
5.1.3	Implementation	47
5.2	Goal Attainment	48
5.3	Threats to Validity	48
6	Related Work	51
7	Conclusions	53
7.1	Summary	53
7.2	Knowledge Gained	54
7.3	Future Work	54
	Appendix A Deliverables	57
	Bibliography	59

1 Introduction

Cloud computing is a type of computing where resources like processing power and storage space can be allocated on demand for applications that are accessible over a network [MG11], usually the Internet. Allocation and deallocation of such resources is characterized by low management costs and can be processed in a short period of time. This low overhead allows cloud providers [LTM⁺11] to use only the resources that are needed and, thus, to save operation costs. Many companies like SAP, Amazon, Google, and Microsoft operate data and computing centers in order to provide cloud platforms for developers, with each platform having its own set of supported programming languages and runtime frameworks. These cloud platforms and frameworks influence performance properties of applications, which makes them relevant for performance engineers.

One approach to analyzing performance is performance prototyping, i.e., the generation of runnable application prototypes that can provide analysis data for concrete operation platforms. Due to the variety of cloud platforms, the generation of these performance prototypes has to be adapted to new platforms. However, current performance prototyping approaches have not yet been evaluated for extensibility, which results in the risk of ineffective adaptations. Therefore, this thesis conducts an extensibility analysis of ProtoCom, a code generator that is part of the performance analysis tool Palladio.

Section 1.1 of this introduction describes the problem statement in more detail and Section 1.2 briefly outlines its solution.

1.1 Motivation

When building software for any cloud platform, engineers face the challenge of measuring quality aspects, e.g., performance. Furthermore, engineers and cloud providers need to adjust the amount of allocated resources to the requirements of the application. These requirements usually vary within certain time intervals because the number of requests is unevenly distributed. For example, the amount of user requests to a web service varies by daytime. Users of a cloud application are sensitive when it comes to performance because too long response times quickly result in dissatisfaction and bad user experience [Nah03]. Therefore, cloud providers try to avoid overload at any rate. A trivial solution to this problem is to provide as many resources as needed to serve all requests at peak times (over-provisioning). The obvious downside of over-provisioning is that most of the resources are unused when demand is low, which results in unnecessary

operation costs. Therefore, performance engineers analyze the performance of an application’s architecture, for instance, by simulating the expected workload in order to make predictions about the resources that are required over time.

One concrete approach to analyzing the performance of an architecture is to create a model of its components and their behavior using an Architecture Description Language (ADL). One example for an ADL is the Palladio Component Model (PCM) [RBB⁺11], which is part of Palladio and used throughout this thesis. Besides the PCM, Palladio consists of the Palladio-Bench, an integrated modeling environment based on Eclipse in which PCM instances can be created and edited. Additionally to the component structure of the system architecture, the PCM describes each component’s performance-relevant behavior using Service Effect Specifications (SEFFs; similar to activity diagrams), the resource environment, the allocation of system components to this environment, and several other performance-relevant aspects like network and middleware. Typically, Palladio is used to analyze an architecture before implementing the actual software and providing hardware resources. Alternatively, Palladio can be used to examine an existing architecture in order to identify potential performance bottlenecks and improvements without interfering with a running system. Furthermore, a sensitivity analysis can be conducted, e.g., to find the best possible resource configuration for a given demand.

Several simulators and solvers [FZI13] can be used to analyze the architecture and derive performance metrics from Palladio models. One of these simulators is ProtoCom [KIT13], a framework that includes load generators and model-to-text (M2T) transformations generating performance prototypes from PCM instances. These performance prototypes are executed on real hardware and consist of a source code representation of the model and a runtime framework that simulates load on the target platform. CPU and hard disk utilization are simulated based on a preceding calibration so that the performance results can be abstracted from the actual hardware that the simulation is performed on. The Palladio-Bench visually presents all results, e.g., response times and resource utilization, in order to make performance bottlenecks and unneeded resources visible for performance engineers.

Currently, ProtoCom generates performance prototypes only for Java SE and, therefore, restricts the performance analysis to a small range of platforms. Even though the generated performance prototype is only used to take measurements and can be viewed as a transparent intermediate step from the model to performance metrics, Lehrig et al. [GL13] have the hypothesis that the choice of language and platform can have a distinct impact on the performance. Thus, it is desirable to facilitate the generation of performance prototypes for as many languages as possible, e.g., Java SE, C#, Python, and JavaScript. For instance, each of these languages requires a different runtime and virtual machine. Furthermore, the large number of cloud platforms and languages requires extensible transformations in order to allow transformation engineers to reuse existing parts that are common to more than one platform or language.

Langhammer et al. [LLK13] describe a means to reuse parts of a transformation that are common to multiple platforms by annotating transformations with target-specific information and introducing an intermediate model between the initial model and its source code representation. However, this approach has only been discussed on a theoretical level and lacks an implementation. Becker presents a conceptual plan for a transformation of PCM instances to Java SE and Java EE in his PhD thesis [Bec08], but this approach lacks automation. Furthermore, since the publication of Becker's thesis, a new version of Java EE was released, which led to a conceptual design of a transformation with focus on Enterprise Java Beans (EJBs) [GL13]. However, this concept misses transformations that would automate the concept. Furthermore, it neglects the generation of Servlets. Some cloud platforms, e.g., the SAP HANA Cloud, do not support the full feature set of EJBs that is necessary for the proposed transformations. As opposed to the communication over RMI-IIOP that is used for EJBs, these platforms require other means of communication, e.g., Servlets and HTTP.

In summary, current concepts either lack automated transformations to Java EE or require features that are missing on certain platforms. Also, the extensibility of performance prototyping approaches such as ProtoCom lacks investigations.

1.2 Solution

To cope with the lack of automated transformations to Java EE as well as the lack of extensibility investigations, the goal of this thesis is to implement M2T transformations of PCM instances to Java EE Servlet performance prototypes and to examine the extensibility of the current ProtoCom transformations during the extension process.

In order to achieve this goal, this thesis follows a simplified Goal Question Metric (GQM) approach. GQM is a goal-oriented top-down method for empirically achieving a goal using metrics and requires the definition of at least one goal, a set of questions that characterize each goal, and a set of metrics that help to answer these questions. The derivation of software metrics using the GQM method is divided into four phases. After the planning phase, in which the fundamentals are discussed, the GQM plan is developed in the definition phase. This plan defines the goal, questions, metrics, and hypotheses. With the simplified approach in this thesis, only one goal regarding the extensibility of ProtoCom is specified. In the subsequent collection phase, the implementation of the M2T transformations is planned and executed. During the implementation, the required data corresponding to the metrics from the GQM plan are collected. Finally, in the interpretation phase, the collected data are analyzed and interpreted in order to assess the extensibility of the ProtoCom transformations. The analysis reveals that the extensibility of ProtoCom can be improved in several areas.

2 Fundamentals

This chapter explains the theoretical and technical fundamentals for this thesis. First, Section 2.1 describes the concept of performance prototyping and outlines how tools like Palladio and ProtoCom can help to automatically generate performance prototypes from models. The ProtoCom transformations that facilitate such automated generations are implemented in the programming language Xtend, which is introduced subsequently in Section 2.2. Afterwards, Section 2.3 describes the Architecture-Centric Model-Driven Software Development (AC-MDSD) approach, which forms the conceptual basis for the transformations. Finally, this chapter introduces the Goal Question Metric (GQM) method in Section 2.4 and the Architecture-Level Modifiability Analysis (ALMA) in Section 2.5. These concepts provide the basis for structuring this thesis and for evaluating and selecting extension scenarios, as discussed in detail in Chapter 3.

2.1 Performance Prototyping

Software engineers and architects aim to analyze quality attributes of a software system, e.g., performance, early in the design process in order to avoid costs for subsequent adjustments. Performance prototyping is an approach that facilitates such analyses by simulating the performance of a software system in a realistic execution environment and under different load levels.

The Palladio Component Model (PCM) is a component-based Architecture Description Language (ADL) that supports engineers in constructing performance prototypes by allowing them to create a formalized model of the components and performance-relevant properties of a software architecture.

The PCM distinguishes between four largely independent modeling tasks, which allows different groups of participants to work on a model simultaneously, as illustrated in Figure 2.1. Component developers specify the components and their internal behavior as well as their provided and required interfaces, while software architects assemble these components in a system and connect them appropriately. Meanwhile, system deployers define the execution environment for the system and assign the components to the available resources. Finally, domain experts specify the workload in the system by modeling user behavior, i.e., arrival rate and input parameter distributions.

After the modeling process is completed, instances of the PCM can be automatically transformed to deployable performance prototypes. This automation reduces the effort for conducting performance simulations and allows engineers to

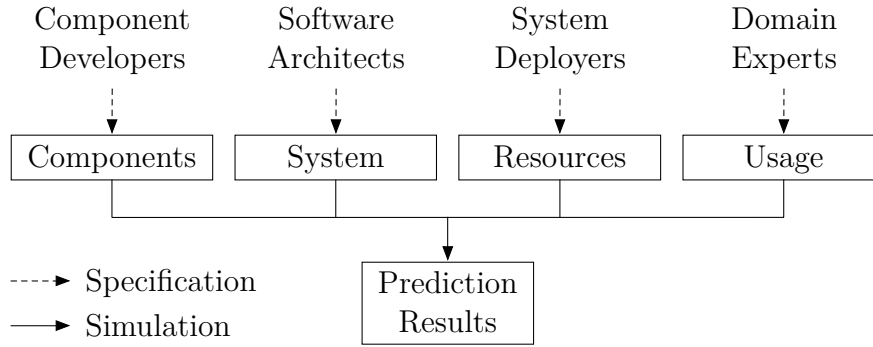


Figure 2.1: Elements of the PCM, cf. [HKR11, p. 29]

efficiently and iteratively analyze how changes of the model and the performance-relevant parameters are reflected in the simulation results.

ProtoCom is part of Palladio and provides such transformations together with a framework for load generation. Although ProtoCom currently supports only transformations to Java SE performance prototypes, it intends to support several other languages (e.g., C#) and platforms (e.g., SAP HANA Cloud and Microsoft Azure) in the future [LLK13] (Figure 2.2). In these transformations, components are mapped to classes whose methods simulate the components' internal behavior using load generators. This functionality is part of ProtoCom's framework and allows to translate hardware-independent resource demands, e.g., CPU utilization and hard disk access, from the PCM instances to actual demands on the executing hardware using a preceding calibration.

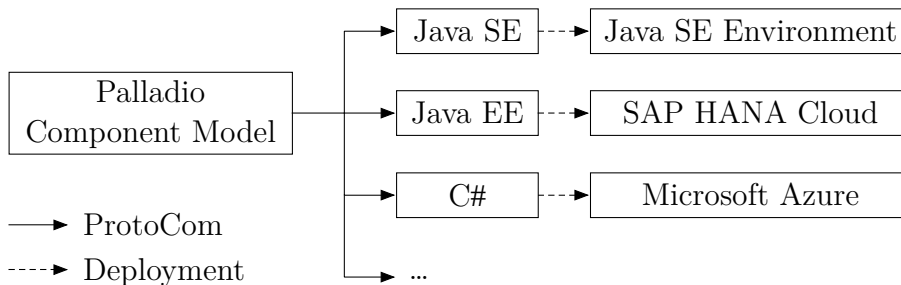


Figure 2.2: Transformation of PCM instances to performance prototypes

2.2 Xtend 2

This section gives an introduction to Xtend 2¹, the implementation language of the ProtoCom transformations. Xtend is based on Java and extends it by various

¹Hereafter referred to as Xtend, without the version number

features, e.g., lambda expressions and operator overloading. Although its syntax is similar to Java, it allows developers to write more concise code. For example, type names can often be omitted because they are inferred from the context at compile time (type inference). Furthermore, Xtend provides a templating system that is especially useful for M2T transformations.

Being closely integrated with Eclipse, Xtend ensures transparent interoperability with Java, as shown in Figure 2.3. Xtend code is compiled to Java code, which is, in turn, compiled to Java bytecode. Therefore, Xtend code can reference other Java code, including compiled libraries, without restrictions. Conversely, Java code that references Xtend code implicitly references the output of the Xtend compiler and, hence, does not need to be aware of Xtend.

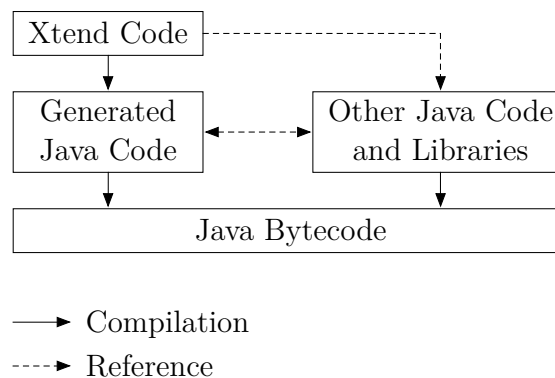


Figure 2.3: Integration of Xtend into the compilation process

The following subsections explain the syntactical and semantical basics of Xtend as well as its templating system. This introduction covers only the most important features required to understand the code listings presented in this thesis. A complete documentation of the language can be found online².

2.2.1 Syntax and Semantics

Most of the syntactical and semantical features of Xtend improve the readability and conciseness of the code, e.g., by reducing redundancy and removing the need for terminating semicolons. However, Xtend is a syntactical superset of Java, thus making all of its features optional. Semantically, Xtend and Java differ the most in the way they handle expressions and statements. In Xtend, the majority of constructs, including statements, are treated as expressions with implicit return types and values, which allows the compiler to perform type inference.

```

1 class CostCalculator {
2   int fixedCosts

```

²<https://www.eclipse.org/xtend/documentation.html>

```
3
4  def void setFixedCosts(int costs) {
5      this.fixedCosts = costs
6  }
7
8  def calculateCosts(int units, int unitCosts) {
9      var variableCosts = units * unitCosts
10     variableCosts + fixedCosts
11 }
12 }
```

Listing 2.1: Class written in Xtend

As shown in Listing 2.1, methods are defined using the **def** keyword. Usually, the return type can be omitted because the last expression in a block determines the block’s type and value. Hence, the return type of the **calculateCosts** method is inferred from the expression in line 10, which is of type **int**. The same principle also applies to **try**, **catch**, **if**, **else**, and **switch** blocks. Similarly, the declaration of variables and constants can take advantage of type inference with the **var** keyword for variables (line 9) and the **val** keyword for constants.

2.2.2 Templates

Although ProtoCom uses many of Xtend’s language features, the main reason why it was selected as implementation language is its templating system. This subsection explains the use of templates on the basis of a small program that transforms class models like the one shown in Figure 2.4 to Java code stubs. The transformation is comprised of three classes (**Class**, **Method**, and **Parameter**), each one being responsible for transforming specific parts of the model.

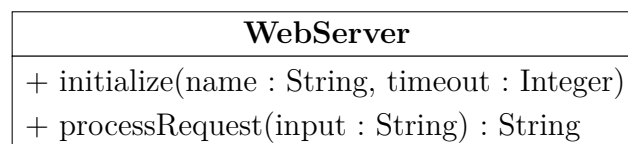


Figure 2.4: Simple class model in UML notation

Templates are similar to strings but can contain basic control flow primitives and expressions that are surrounded by guillemets (« and ») and evaluated at runtime. The templates themselves are enclosed by triple single quotes ('''') and can span multiple lines.

Listing 2.2 shows the part of the transformation that is responsible for generating the code of a class. The **generate** method generates the class declaration and iterates over the list of methods in order to delegate the transformation to the respective **Method** instances (lines 8 to 10). **FOR** loops allow to specify an optional

separator string that is inserted between every two iterations. In this case, an empty line is added between the methods.

```

1 class Class {
2   String name
3   List<Method> methods
4   ...
5   def generate() {
6     '''
7     class «name» {
8       «FOR m : methods SEPARATOR "\n"»
9       «m.generate»
10      «ENDFOR»
11    }
12    '''
13  }
14 }

```

Listing 2.2: The code for transforming a class

Similarly, methods are transformed by inserting their type and name, delegating the transformation of parameters to the respective **Parameter** instances, and separating each pair of parameters with a comma, as demonstrated in Listing 2.3. The **generate** method of the **Parameter** class works analogously to the other transformations and is, therefore, not further explained here.

```

1 class Method {
2   String type
3   String name
4   List<Parameter> parameters = new ArrayList<Parameter>
5   ...
6   def generate() {
7     '''
8     public «type» «name»(«FOR p : parameters SEPARATOR ", "»«p.generate»
9       ↪ «ENDFOR») {
10      // Method body
11    }
12    '''
13  }
14 }

```

Listing 2.3: Code for transforming a method

A call to the **generate** method of a **Class** instance that stores the data of the model shown in Figure 2.4 returns a string containing the corresponding Java code stubs (Listing 2.4).

```

1 class WebServer {
2   public void initialize(String name, Integer timeout) {
3     // Method body
4   }
5
6   public String processRequest(String input) {
7     // Method body
8   }
9 }

```

Listing 2.4: Output of the transformation

2.3 Architecture-Centric MDSD

Model-Driven Software Development (MDSD) is a development approach that aims to increase development efficiency and software quality through automated transformations of models to executable code [SVC06]. In order to abstract from technical details and actual programming languages, these models are constructed in languages geared towards the domain of the application to be developed. Such languages are thus called domain-specific languages (DSLs). An example for a DSL in the domain of performance prototyping is the PCM introduced in Section 2.1.

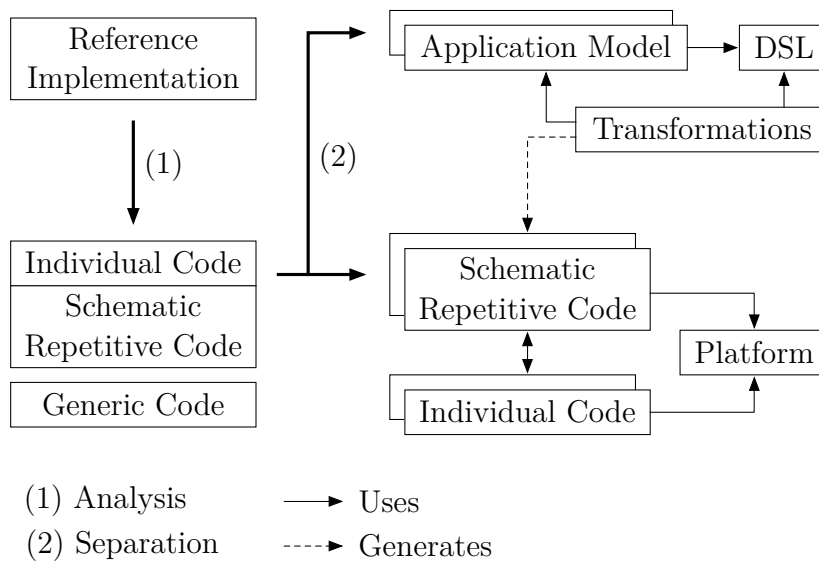


Figure 2.5: Elements of AC-MDSD, cf. [SVC06, p. 15]

Architecture-Centric Model-Driven Software Development (AC-MDSD) is a special form of this concept that focuses on the generation of architectural code for an application [SVC06]. Large parts of this architectural code, e.g., for an MVC pattern, follow a certain scheme and have to be re-written each time the architecture

is implemented. This type of code is also called infrastructure code or schematic repetitive code and can be generated automatically from models using AC-MDSD. However, generating code using this approach only provides a skeleton containing the architectural infrastructure. Individual parts of the application, e.g., business logic, have to be supplemented manually. Figure 2.5 illustrates the elements of AC-MDSD and shows how automated transformations for schematic repetitive code can be derived from a reference implementation.

The reference implementation represents a concrete implementation of the architecture, which is analyzed in order to separate its code into three categories: generic code, individual code, and schematic repetitive code. Generic code is independent of the actual implementation, e.g., frameworks and third-party libraries, and can be reused without changes. Individual code contains the business logic and is, therefore, specific for the examined implementation. Schematic repetitive code is also specific for the implementation but follows a scheme that makes it eligible for automated transformations. As shown on the righthand side of the diagram, the models of the application and the code that they are transformed to can be considered separately. The application models constructed in the DSL are transformed to schematic code that is supplemented by manually written individual code. The DSL, transformations, and platform code only need to be created once for each architecture and can be reused for arbitrary application models.

2.4 Goal Question Metric (GQM) Method

In software development, engineers can take measurements as a means for feedback and evaluation. However, in order to be effective and interpreted correctly, these measurements have to be organized and defined in a goal-oriented, top-down fashion [BCR94]. The Goal Question Metric (GQM) method described in this section is an approach for structuring the measurement process and hierarchically deriving questions and metrics from goals. The method, as illustrated in Figure 2.6, consists of four phases.

In the planning phase, preparatory work and project planning is performed. Afterwards, during the definition phase, goals, questions, metrics, and hypotheses are specified in the GQM plan. Starting with one or more goals, the GQM plan refines each goal into a set of questions. For example, the goal “Analyze the extensibility of an M2T transformation” can be refined into questions like “How much effort is required for implementing an additional output language?” For each of these questions, the GQM plan identifies metrics that are used for measurement, e.g., working hours and lines of code. Additionally, the plan contains hypotheses that state the expected outcome of the measurements. In the subsequent collection phase, the data are collected based on the metrics specified in the GQM plan.

The purpose of the final phase, the interpretation phase, is to evaluate the collected data and to compare the actual outcome of the measurements with the hypotheses of the GQM plan. In order to achieve meaningful results, the raw

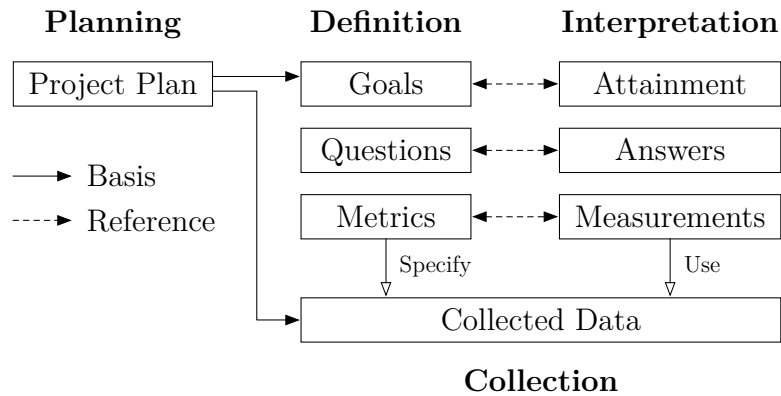


Figure 2.6: Phases of the GQM method, cf. [vSB99, p. 22]

measurement data has to be interpreted in terms of the goals. Hence, the interpretation of measurements is conducted bottom-up by answering the questions based on the collected data and then using these answers to evaluate the attainment of the goals.

2.5 Architecture-Level Modifiability Analysis

When designing architectures, software architects have to make far-reaching decisions that determine the architectures' quality attributes. Analyzing architectures early in the design process, ideally before the implementation, allows for easier adjustments and risk reductions, whereas changes done later in the life cycle can be costly and time-consuming.

The Architecture-Level Modifiability Analysis (ALMA) method described here is a scenario-based method for evaluating software architectures in terms of modifiability [BLBvV04] (including its subtype extensibility [FCH⁺08]). The procedure is similar to the Architecture Tradeoff Analysis Method (ATAM), an approach for evaluating software architectures relative to quality attribute goals, e.g., security and performance [KKC00]. However, ALMA focuses only on modifiability instead of examining tradeoffs between different quality attributes. An analysis following the ALMA method is conducted in four steps.

First, the goal of the analysis is specified. ALMA distinguishes between three different types of goals: maintenance cost prediction (estimating the effort that is required for implementing changes), risk assessment (identifying points in the architecture that are inflexible for changes), and architecture selection (comparing two or more architectures). Each of these goals leads to a different focus regarding the selection of scenarios in the third step of the analysis.

In the second step, information about the architecture is collected in order to allow analysts to evaluate the impact of the scenarios. This information includes the components of the system and their relationships as well as their interactions with the system's environment.

In the third step, analysts and stakeholders elicit scenarios that facilitate the modifiability analysis. For example, the scenario "Add C# as additional output language to an M2T transformation" can be used to examine the extensibility of the part of the architecture that is responsible for M2T transformations. These scenarios can either be derived from the information gathered in the second step (top-down) or suggested by stakeholders (bottom-up). If the analysis follows the bottom-up approach, a brainstorming involves different groups of stakeholders, e.g., software architects, developers, product managers, and users, in the scenario elicitation. The benefits of this cooperation are increased communication between stakeholders and a set of scenarios in which different perspectives are considered.

In the final step, analysts and software architects evaluate the impact of the scenarios on the architecture and the effort that is needed to implement them. For that purpose, directly and indirectly affected components are identified for each scenario. For example, a change in one component may recursively cause changes in other components (ripple effects), which can adversely affect the modifiability of the system. Finally, the results for each scenario are interpreted in terms of the goal set in the beginning of the analysis.

In this thesis, a slightly adapted version of the ALMA method is used. According to the goal of this thesis, the analysis focuses only on extensibility and considers transformation engineers as the only stakeholders. Instead of estimating maintenance costs, it is primarily used for evaluating the importance and complexity of possible extension scenarios. The results of the analysis are then used as a basis for selecting the scenarios to implement, as discussed in Chapter 3.

3 Definition Phase

This chapter lays the groundwork for the collection phase and specifies the GQM plan for this thesis, starting with the definition of the goal in Section 3.1. Afterwards, Section 3.2 describes the architecture of ProtoCom and the performance prototypes it generates. On that basis, the Architecture-Level Modifiability Analysis method is used in Section 3.3 to identify and evaluate extension scenarios that are suitable for an extensibility analysis of ProtoCom. Finally, Section 3.4 defines questions, hypotheses, and metrics for these extension scenarios, which—together with the previously defined goal—result in the GQM plan for this thesis.

3.1 Goal Definition

The goal of this thesis is to analyze the extensibility of ProtoCom by extending it on the basis of extension scenarios and taking measurements during their implementation. Table 3.1 defines this goal according to the template provided by van Solingen and Berghout [vSB99].

Table 3.1: Definition of the goal, cf. [vSB99, p. 51]

<i>Analyze</i>	the architecture of ProtoCom
<i>For the purpose of</i>	maintainability assessment
<i>With respect to</i>	extensibility
<i>From the viewpoint of</i>	transformation engineers
<i>In the context of</i>	performance prototype transformations

3.2 ProtoCom Architecture

ProtoCom follows the AC-MDSD approach to transform PCM instances to performance prototypes, as shown in Figure 3.1. The ProtoCom transformations receive a PCM instance as input and use the information in the model to generate the code of the performance prototype. The transformations have a modular design in which the generation of language constructs, e.g. classes, methods, and generic XML files, is separated from the generation of technology-specific code, e.g., calls to framework functions. The traversal component shown in the diagram combines the appropriate language and technology components, depending on the desired configuration. The framework (which corresponds to generic platform code in

AC-MDSD) is referenced by the final output of the transformation and bundled with the performance prototype during deployment.

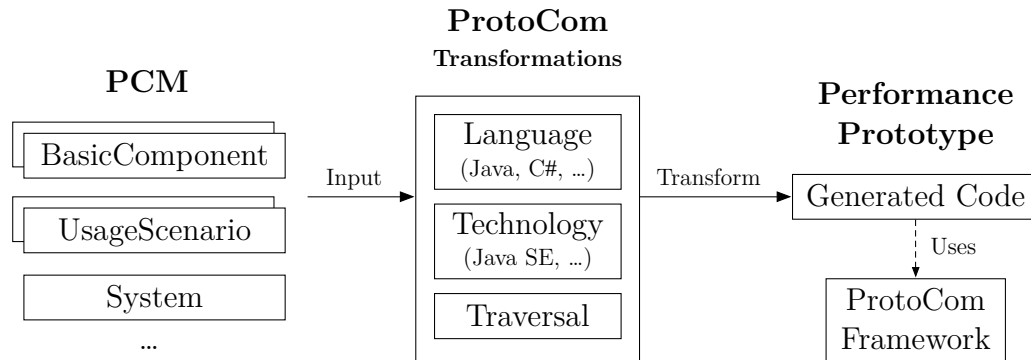


Figure 3.1: Overview of the ProtoCom transformations

Figure 3.2 shows the components of the Java SE ProtoCom framework and a generated performance prototype. The entry point of the performance prototype is represented by the *Main* component. It receives all user interaction by providing a command-line user interface and launches the requested parts of the performance prototype, i.e., RMI registry, resource containers, system, and usage scenarios.

The remaining components of the prototype are directly derived from the input model, whereby all PCM components and interfaces are subsumed under the *Repository* component. Similarly to the *Main* component, the *Allocation*, *Resource Environment*, and *Usage Scenarios* components have counterparts in the framework that provide common, model-independent functionality. For example, the *Abstract Resource Environment* component manages the calibration of the CPU and HDD, whereas the *Resource Environment* component provides the capabilities of the processing resources specified in the model.

Besides these direct interfaces for the generated code, the framework also contains different calibration methods (*Resource Strategies*) and means of collecting and storing measurement results (*Experiment Manager*). The *RMI Registry* component provides RPC functionality for launching the different parts of the prototype on individual hosts, which allows performance engineers to take measurements in a distributed system.

3.3 Extension Scenarios

According to the Architecture-Level Modifiability Analysis method described in Section 2.5, several possible extension scenarios were discussed in a brainstorming. Four software engineers from the Software Engineering Research Group at the University of Paderborn who are involved with ProtoCom and model-driven software development participated in this discussion, which resulted in a set of

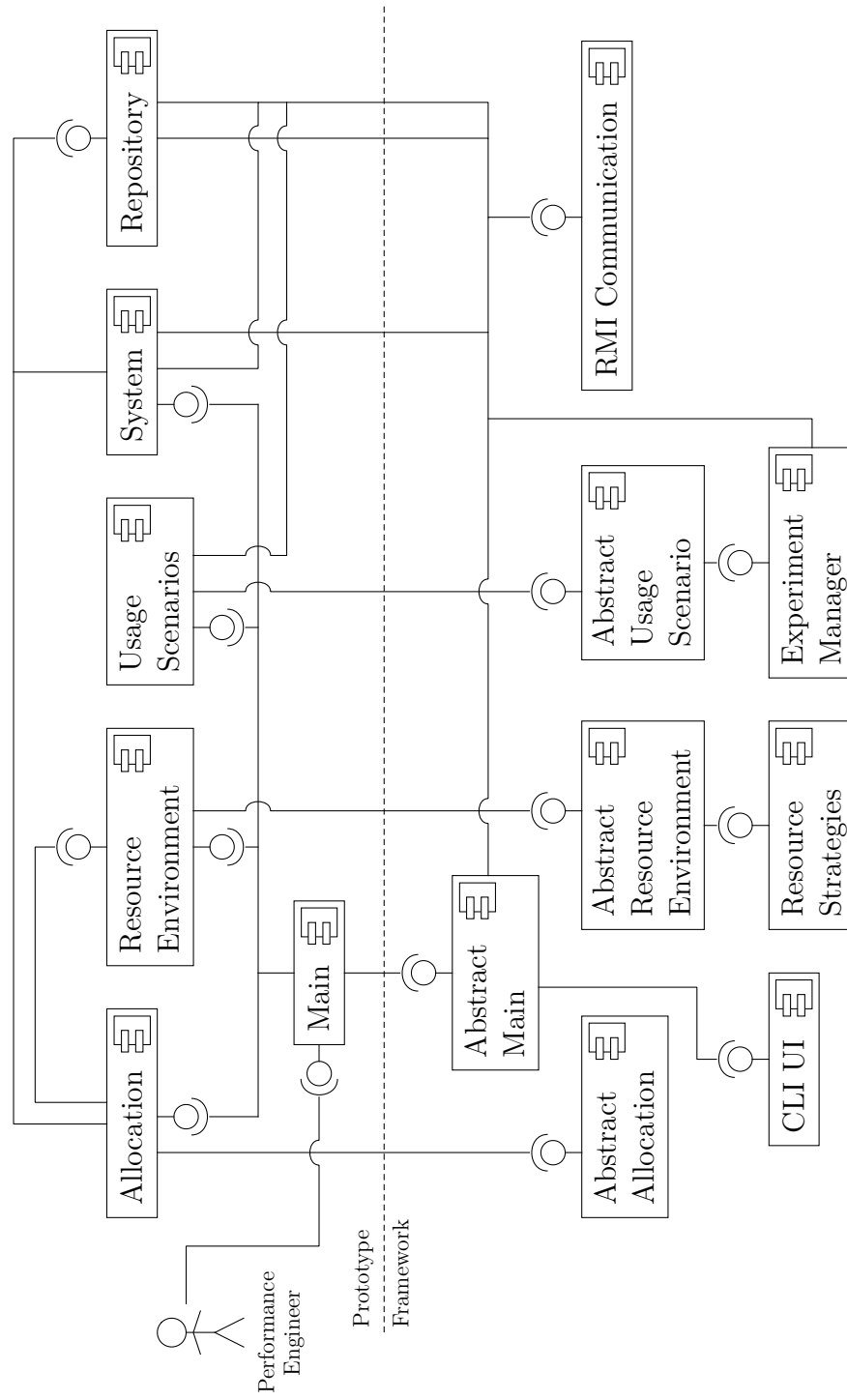


Figure 3.2: Components of Java SE performance prototypes and the ProtoCom framework

transformation and framework extensions. The document describing the starting points for the brainstorming can be found in the deliverables listed in Appendix A.

This section presents the results by describing a set of high-level extension scenarios in Section 3.3.1 and discussing their advantages and disadvantages for an extensibility analysis of ProtoCom. Afterwards, it selects the high-level scenario that promises the best tradeoff between implementation complexity and significance in terms of extensibility. Finally, it refines this scenario into smaller, architecture-oriented extensions, as described in Section 3.3.2 and Section 3.3.3.

3.3.1 High-Level Extension Scenarios

Applying the AC-MDSD approach to ProtoCom yields several starting points for defining high-level extension scenarios. Regarding the elements of AC-MDSD, there are three extension scenarios that affect ProtoCom, as shown in Table 3.2.

Table 3.2: High-level extension scenarios for ProtoCom

Scenario 1	Add a new target language, e.g., C#
Impacts	Transformations (high), Framework (high)
Hypothesis	Hard to accomplish because existing Java parts of the transformations cannot be reused

Scenario 2	Add features to the PCM, e.g., using PCM Profiles
Impacts	PCM (low), Transformations (low), Framework (low)
Hypothesis	Easy to accomplish because only small changes in the affected components are required

Scenario 3	Add a new target technology for an existing language
Impacts	Transformations (medium), Framework (high)
Hypothesis	Easy to accomplish because existing Java parts of the transformations can be reused and because ProtoCom was designed to allow this kind of extension

First, a new output language can be added to ProtoCom. This extension allows for a complete extensibility analysis of ProtoCom because all parts of the transformations as well as the framework are affected. Adding a new language requires new transformations and framework components to be developed from the ground up, allowing the implementation to take only limited advantage of existing parts. However, reusing such existing parts can be important for the analysis in order

to evaluate their extensibility. Furthermore, this scenario is expected to require large implementation effort, due to the high impact on the affected components.

Second, the PCM can be extended, e.g., to support annotations that influence the transformations. Possible extensions include the annotation of components such that the target language and technology can be selected individually for each component. Furthermore, connectors could be annotated to select the communication protocol between two components and to enable end-to-end encryption. These modifications of the PCM affect the transformations and the framework. For example, adding encryption capabilities to the communication between components requires the implementation of key exchange and encryption mechanisms in the framework, as well as appropriate method calls in the code output of the transformations. This kind of extension is presumably easy to implement because it requires only small changes in each of the components involved. For example, adding encryption capabilities leaves most of the framework components untouched. However, such isolated extensions make it difficult to give generally valid statements about the extensibility.

Third, a new target technology, e.g., Java EE Servlets, can be added for an existing language. Compared to the first scenario, this extension has a lower impact on the transformations because it can reuse the language-specific parts of ProtoCom. This scenario is supposed to be easy to implement because ProtoCom is designed to allow this kind of extension through its modular design, as illustrated in Figure 3.1.

However, ProtoCom currently lacks an evaluation of these extensibility characteristics. Therefore, this thesis pursues the implementation of the third high-level extension scenario with Java EE Servlets as target technology.

The following two sections refine this scenario by breaking it down into smaller scenarios that can be implemented individually.

3.3.2 Transformation Extensions

The extensions of the transformations are categorized by PCM entities, as shown in Table 3.3. This thesis focuses on a representative subset of the entities supported by Java SE ProtoCom by considering only the most essential entities for transforming runnable performance prototypes. However, this subset still ensures that generally valid statements can be made regarding the extensibility of the ProtoCom transformations. For instance, the *System* and *CompositeComponent* entities, as well as the *OperationInterface* and *InfrastructureInterface* entities, have similar characteristics. Therefore, the extensibility analysis can exclude the transformations of some entities. The *System* and *OperationInterface* entities are included because they are obligatory for valid PCM instances.

The *UsageScenario* entity is also excluded in the automated transformations. Instead, a proof of concept is provided for one specific PCM instance. The actual transformation for usage scenarios of arbitrary PCM instances remains as future work.

Table 3.3: Implemented entities, compared to Java SE ProtoCom

Included entities	Excluded entities
OperationInterface	InfrastructureInterface
BasicComponent	CompositeComponent
System	UsageScenario
ResourceEnvironment	
Allocation	

3.3.3 Framework Extensions

Each of the following component diagrams illustrates the changes of the architecture—compared to the components shown in Figure 3.2—that result from the extensions in the framework.

First, the *RMI Communication* component is replaced by the *HTTP Communication* component (Figure 3.3) because Java EE cloud platforms only allow HTTP communication across different hosts. Therefore, Java RMI cannot be used on such platforms. The *HTTP Communication* component contains functionality for the endpoints of communicating system and component entities, as explained in detail during the collection phase. Furthermore, this component provides an HTTP-based registry—similar to the Java RMI registry—as a central registration point for system and component entities. This registry and the communication protocol are explained in Section 4.2.3 and Section 4.2.4, respectively.

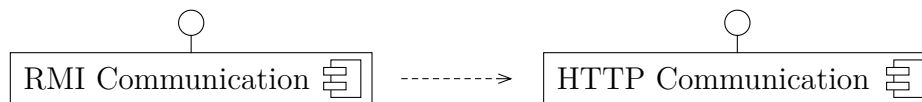


Figure 3.3: Communication extension

Second, the *CLI UI* component is replaced by the *Web UI* component (Figure 3.4). It provides an HTML user interface by which performance engineers can launch individual parts of the performance prototype, similar to the CLI user interface of Java SE ProtoCom. However, an HTML user interface is required because Java EE cloud platforms do not provide console-level access to cloud applications.

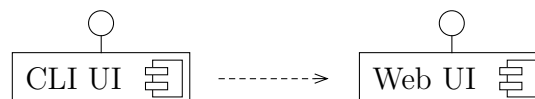


Figure 3.4: User interface extension

Third, all components involved in the execution of usage scenarios are replaced by the *Web Load Driver* component (Figure 3.5), which serves as an interface for external load generators. As explained in Section 3.3.2, transformations of usage

scenarios are not considered in this thesis. Instead, a proof of concept is provided for the external HTTP load generator Apache JMeter¹. This load generator is controlled by performance engineers when taking measurements and issues HTTP requests at the *Web Load Driver* component, which are then delegated to the appropriate components inside the system.

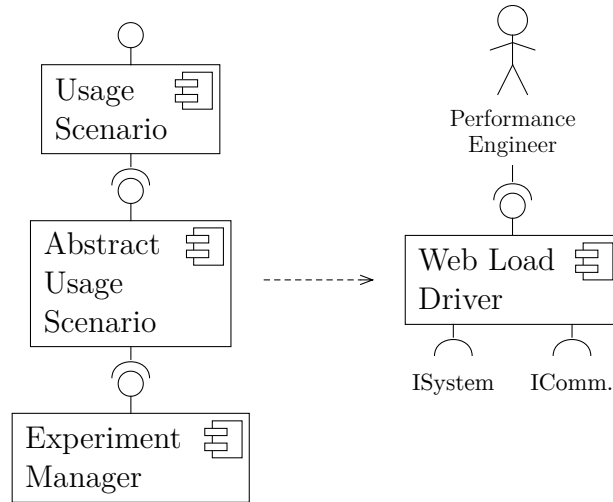


Figure 3.5: Usage scenario extension

3.4 Questions, Hypotheses, and Metrics

Resulting from the separation of transformation and framework extensions, this thesis aims to answer the following two questions derived from the goal: “*How extensible are the ProtoCom transformations?*” and “*How extensible is the ProtoCom framework?*”.

3.4.1 Transformations

As described in Section 3.3.1, the ProtoCom transformations are designed to allow extensions for other target technologies. Thus, it is expected for these transformations to have largely good extensibility characteristics and to require only minor or no changes in the language-specific parts. Furthermore, the extensions are presumably easy to integrate into existing structures because ProtoCom provides interfaces for this purpose. However, the documentation of the transformations seems to be incomplete and—to some extent—of low quality, which may have a negative impact on the extensibility.

¹<http://jmeter.apache.org>

3.4.2 Framework

Regarding the ProtoCom framework, a first analysis showed that the functionality is heavily technology-oriented and monolithic. As illustrated in Figure 3.2, a majority of the components in both the framework and the generated performance prototype directly rely on the *RMI Registry* component. However, this component is specific for Java SE and cannot be reused in many other runtime environments, e.g., the SAP HANA Cloud. Such environments require different means of communication. Overall, the orientation of the framework towards Java SE is visible in almost each of its components, due to the use of direct file system access and command-line argument handling. Therefore, the framework extensions are expected to require large effort for both separating and reimplementing required functionality.

3.4.3 Metrics

This section describes the metrics used for the measurements during the collection phase. The actual measurements for these metrics are discussed during the interpretation phase in Chapter 5.

Non-Commenting Source Statements (NCSS) This thesis uses the NCSS² metric to measure the size of existing code and the amount of code added during the implementation in order to quantify the required effort. NCSS are similar to LoC (Lines of Code) but aim to provide a more accurate representation of code sizes. For example, two statements written in one line would result in one LoC but two NCSS. Hence, this metric eliminates size variations resulting from coding style and takes the actual code into consideration. Due to the lack of code metric tools for Xtend, this thesis uses JavaNCSS to take NCSS measurements for the Java output generated by the Xtend compiler. However, the size of the generated code is expected to positively correlate with the size of the Xtend code.

Implementation Time In order to further quantify the implementation effort, this thesis measures the time required by the author to implement the transformation and framework extensions. Additionally, this metric is used to identify extensibility weaknesses by comparing the time measurements with the implemented NCSS.

Abstractness The number of interfaces/abstract classes (AC) and concrete classes (CC) as well as the rate of abstract classes in a package can be an indicator of its extensibility [Mar03]. For example, a low abstractness indicates a high coupling and, therefore, bad extensibility. The abstractness A of a package is defined as the ratio of abstract classes to the total number of classes, as shown in Equation 3.1.

²<http://www.kclee.de/clemens/java/javancss/#specification>

The possible values of A range from $A = 0$ (completely concrete package) to $A = 1$ (completely abstract package).

$$A = AC/(AC + CC) \quad (3.1)$$

Couplings and Instability Couplings and instability indicate how a package reacts to external changes [Mar03] and are measured on a per-package level, similar to the abstractness metric.

There are two types of couplings: afferent couplings (Ca) and efferent couplings (Ce). The number of afferent couplings is an indicator of a package's responsibility and equates to the number of external dependencies on the considered package. Analogously, the number of efferent couplings is an indicator of a package's independence and equates to the number of dependencies on external packages. The instability I of a package is defined as the ratio of efferent couplings to the total number of couplings, as shown in Equation 3.2. The possible values of I range from $I = 0$ (completely instable package) to $I = 1$ (completely stable package).

$$I = Ce/(Ce + Ca) \quad (3.2)$$

A high instability indicates that a package can be impaired by external changes to other packages. The required adjustments that result from this impairment can hurt the extensibility.

Documentation Although the documentation of classes and methods is only indirectly related to the extensibility of the architecture, it can have an important impact on the transformation engineer's ability to understand and properly reuse existing functionality [GFA09]. Hence, this thesis measures the number of classes and methods in the *Language* component documented with Javadoc. Ideally, all code constructs, e.g., classes and methods, should provide documentation that is sufficient for understanding their meaning. However, overly extensive documentation can also have negative impacts due to longer reading time.

4 Collection Phase

This chapter presents the implementation of the extensions specified in Chapter 3. First, it explains how the entities and concepts of the PCM are mapped to Java and the Servlet target technology in Section 4.1. Afterwards, Section 4.2 presents a reference implementation, which—following the AC-MDSD approach—consists of a PCM instance and the respective performance prototype. This reference implementation is then used in Section 4.3 to build automated transformations for arbitrary PCM instances.

4.1 Mapping the PCM to Java

When generating code from PCM instances, the entities and concepts of the source model have to be mapped to constructs of the target language. This section explains how these mappings are accomplished.

4.1.1 Language Constructs

Although both the PCM and Java have similar constructs, e.g., interfaces and components/classes, they differ in expressiveness. Therefore, a one-to-one mapping is sometimes impossible. Table 4.1 lists the PCM entities and concepts regarded in the extension scenarios together with their respective Java mapping.

Table 4.1: Mapping of PCM entities and concepts to Java, cf. [GL13]

PCM Entity/Concept	Java
Interface	Interface
Component	Component class
Provided role	Port class
Required role	Context class
System	System class
Assembly context	Component class instance
Call action	RPC over HTTP
Control flow	Control flow
Resource environment	Resource environment class
Allocation	Allocation class
Usage scenario	[External]

PCM interfaces are directly mapped to Java interfaces, while components, systems, resource environments, and allocations are mapped to Java classes, with their methods and logic being derived from the data in the model. However, the relationship of components and interfaces (provided and required roles) needs additional classes to be generated, as explained in Section 4.1.2.

The instantiation of component classes is handled by the transformed system class. At this point, it should be mentioned that the implementation restricts the set of valid input models—similar to Java SE ProtoCom—by assuming that a component can exist at most once inside a system. In fact, the PCM assigns each component instance a unique assembly context identifier by which it can be referenced, thereby allowing multiple instances of the same component to coexist inside a system. Implementing this feature for the Java EE Servlet transformations remains as future work.

Finally, local control flow and resource demands from the Service Effect Specifications (SEFFs) are transformed to Java control flow and calls to framework methods.

4.1.2 Proxy and Context Patterns

Provided roles (i.e., implemented interfaces) of a component cannot be mapped directly to Java, because a component can provide multiple roles, with the possibility of their interfaces to declare methods with identical signatures [Bec08, p. 178]. However, methods of different provided roles—despite identical signatures—provide different implementations. If the transformations would directly translate PCM interfaces to Java interfaces and components to classes implementing these interfaces, the methods with identical signatures would share the same implementation.

The ProtoCom transformations deal with this issue by generating proxy classes (ports) that delegate method calls to the correct implementation according to the proxy pattern (cf. [GHJV95]), as illustrated in the top half of Figure 4.1. For each role that a component provides, a Java port class implementing the corresponding interface is generated. Therefore, PCM interfaces can be mapped directly to Java interfaces. All multiplexing and the delegation of method calls is handled by the port classes and the component itself. The methods of this component are prefixed by the respective port name. For example, a component providing two roles that both contain an **operation** method is transformed to a class with the two methods **Port-1_operation** and **Port-2_operation**, thereby resolving the issue of identical method signatures.

In addition to allowing a correct mapping of provided roles to Java, port classes have several other advantages. For example, method calls can be intercepted for logging and response time measurement. Furthermore, ports can implement RPC protocols to eliminate the locality of direct method calls, allowing components to interact with each other in a distributed system. Ports generated by Java SE ProtoCom use Java RMI for this kind of communication, whereas the implementation

presented in this thesis uses a custom, HTTP-based RPC protocol, as described in Section 4.2.4.

In order to provide this kind of decoupling not only for provided roles but also for required roles (i.e., references to components implementing required interfaces), ProtoCom uses the context pattern (cf. [SVC06]), as illustrated in the bottom half of Figure 4.1. For each component class, a corresponding context class responsible for managing the required roles is generated. This context class provides a getter method for each required role, which returns a reference to a port object implementing the required role's interface. That way, components are not required to know which objects implement these interfaces and, thus, do not have to instantiate them explicitly. Instead, the instantiation of these objects and their connection to the appropriate contexts is handled by the system during startup. In this case, the system serves as a configuration module that determines the dependencies among components (dependency injection). Furthermore, the context pattern allows components to be swapped individually in the context or altogether by replacing the entire context.

4.2 Reference Implementation

This section presents the reference implementation from which the general transformations are derived. First, it describes the elements of the PCM reference instance in Section 4.2.1 and its transformation in Section 4.2.2, followed by the functionality of the HTTP registry in Section 4.2.3, the communication protocol in Section 4.2.4, and the web user interface in Section 4.2.5.

4.2.1 PCM Reference Instance

The reference model consists of a minimal set of components and a minimal system setup (cf. [GL13]) but still ensures that the manual transformations cover all PCM entities and concepts explained in Section 4.1.1. The component repository is shown in Figure 4.2 and contains the two components *Alice* and *Bob* with their corresponding provided interfaces *IAlice* and *IBob*. Additionally, the *Alice* component requires methods from the *IBob* interface.

For the purpose of this reference implementation, the components do not consume any CPU or HDD resources but instead perform minimal communication, as illustrated in Figure 4.3. The **callBob** method executes an external call to the **sayHello** method of a component providing the *IBob* interface and the **sayHello** method implemented by the *Bob* component does not perform any operation. More complex SEFFs with branches, loops, and resource demands can directly be reused from Java SE ProtoCom.

Instances of the components are created and connected by the system shown in Figure 4.4. The *Alice* and *Bob* components are instantiated once and connected

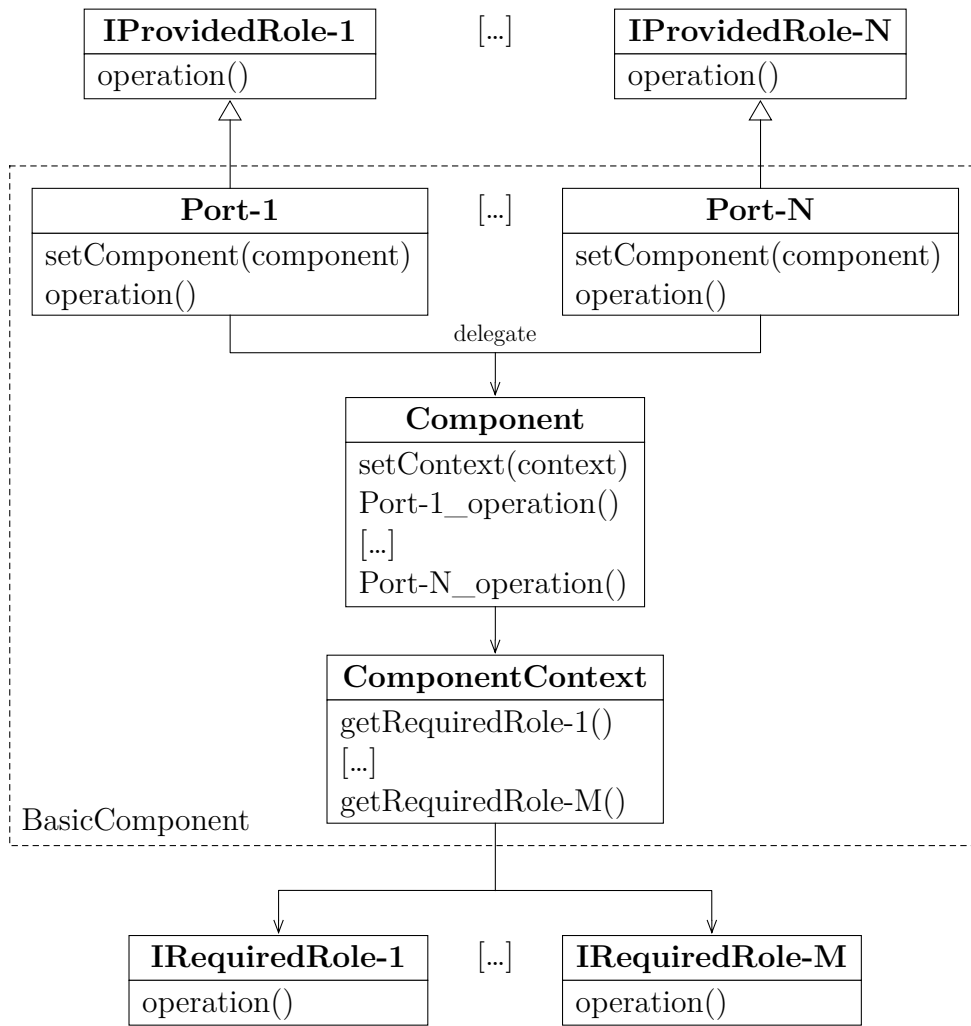


Figure 4.1: Proxy and context patterns in ProtoCom

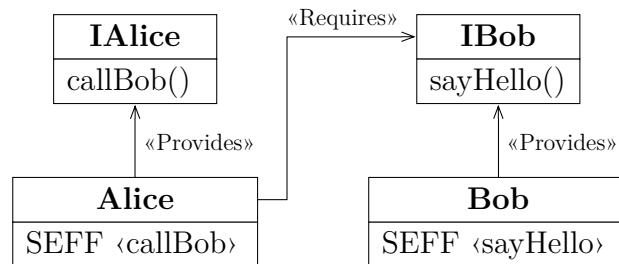
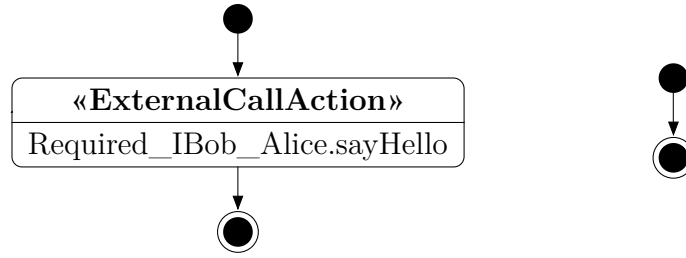


Figure 4.2: Component repository of the reference implementation

Figure 4.3: SEFFs of the `callBob` (left) and `sayHello` (right) methods

according to their provided and required roles. The *Assembly_Bob* assembly context satisfies the required *IBob* role of the *Assembly_Alice* assembly context and the provided *IAlice* role of the *Assembly_Alice* assembly context remains as the provided role of the system by which external usage scenarios can initiate requests.

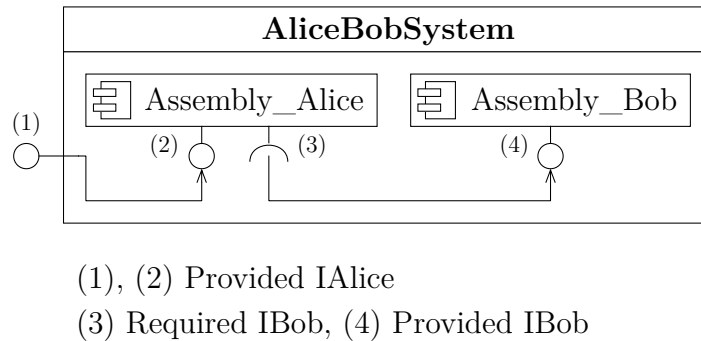


Figure 4.4: System of the reference implementation

The resource environment is omitted here because its transformation is reused from Java SE ProtoCom. Furthermore, the automated transformation of usage scenarios remains as future work, as discussed in Section 3.3.3.

4.2.2 Transformed Model

The class diagram in Figure 4.9 shows the Java transformation of the previously described PCM instance. The **Alice**, **IAlicePort**, and **AliceContext** classes are generated from the *Alice* component, according to the patterns described in Section 4.1.2. The same applies to the *Bob* component. However, the generated **Bob** class does not require a dedicated context class, because it has no required roles.

The **Registry** class is the central contact point for component and system ports. As with Java RMI, it allows an object to be registered under a unique name. Objects at other hosts can then use this name to look up a remote reference of the registered object. The **lookup** method returns a stub that delegates any method calls invoked on it to the actual remote instance of the object. The

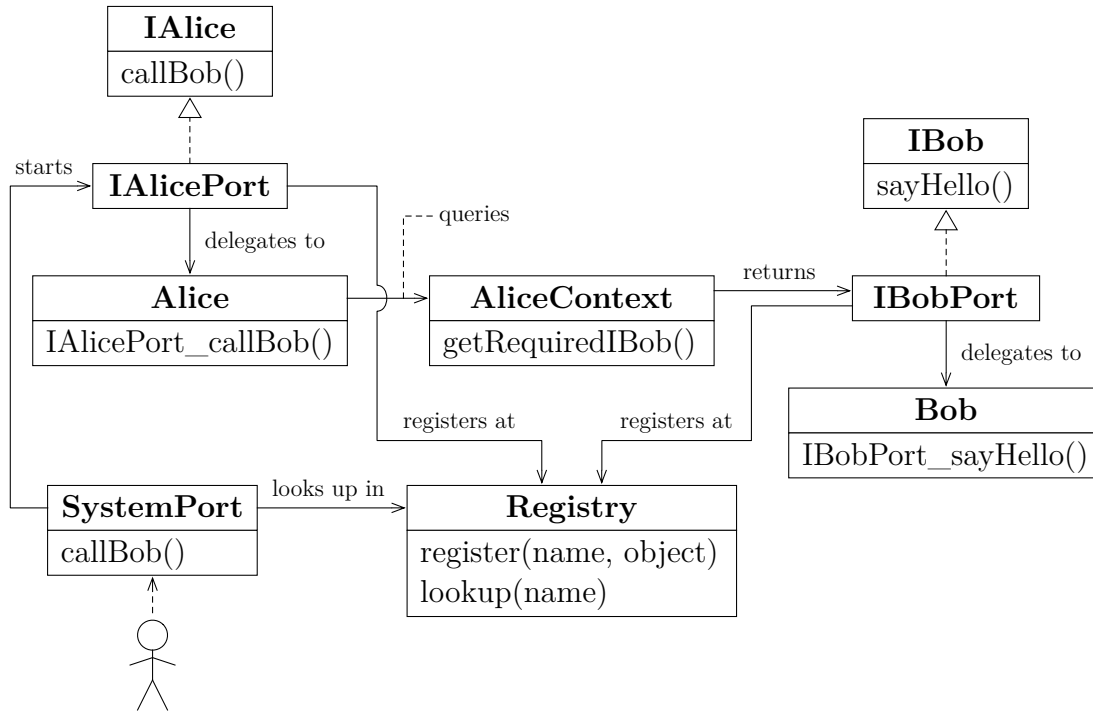


Figure 4.5: Class diagram of the transformed reference model

functionality of the **Registry** class is explained in detail in Section 4.2.3. During their initialization, the port objects of the components register themselves with the names of their implemented interfaces, so that they can be found by each other and by the system.

For simplicity, the generated **System** class is left out in this diagram because its sole purpose is to instantiate component classes and to connect them to the appropriate contexts during startup by looking up references in the registry. It is not involved with any other functionality during runtime.

Performance engineers interact with the **SystemPort** class via an external load generator, e.g., Apache JMeter, that executes an arbitrary usage scenario. This system port implements the methods of the system's provided role (*IAlice*) and delegates the according method calls to the **IAlicePort** object that it previously retrieved through a registry lookup. Afterwards, the **IAlicePort** object forwards the method call to the **Alice** object, which retrieves an object implementing the **IBob** interface through its context and executes the transformed SEFF by calling the `sayHello` method of this object. Finally, the **IBobPort** object forwards the method call to the **Bob** object.

4.2.3 HTTP Registry

The entire communication between components and the system is realized with Servlets, using HTTP as application protocol and JSON¹ as underlying data format. Servlets are Java classes that handle client requests inside a web server. Instances of these classes are managed by a Servlet container, e.g., Apache Tomcat², which is also used in the SAP HANA Cloud. Since the components and the system can reside at different hosts, their ports are implemented as Servlets, each with its own URL used for receiving method invocations.

In order that the components and the system can determine each others hosts, the application requires a central registry. For that purpose, the framework provides a registry consisting of two parts: a remote registry—implemented as a Servlet—that constitutes the central storage for all registered port Servlets and a local registry represented by the **Registry** class that functions as a proxy for its remote counterpart.

In contrast to Java RMI, this registry does not accept object instances for registration but instead relates the unique registration name to a Java interface and the URL of a Servlet implementing that interface. Listing 4.1 shows the method call that registers the **IAlicePort** Servlet. The first parameter specifies the name used for registration and the second parameter specifies the interface that the Servlet implements. The concatenation of the third and fourth parameter forms the URL of the Servlet.

```

1 Registry registry = Registry.getInstance();
2 String location = "http://localhost:8080/Prototype";
3 registry.register("IAlice", IALice.class, location, "/IAlicePort");

```

Listing 4.1: Method call for registering the **IAlicePort** Servlet

The local registry serializes that information to the JSON string shown in Listing 4.2 and forwards it to the remote registry using a GET request. Finally, the remote registry deserializes the received JSON string and stores it for later lookup access.

```

1 {
2   "name": "IAlice",
3   "interface": "repository.alice.IALice",
4   "location": "http://localhost:8080",
5   "path": "/IAlicePort",
6 }

```

Listing 4.2: JSON representation of a registration request

¹<http://json.org>

²<http://tomcat.apache.org>

The **lookup** method receives the name of a registered Servlet as parameter and returns a stub object that implements the interface specified during the registration, as shown in Listing 4.3.

```

1 Registry registry = Registry.getInstance();
2 IAlice alice = (IAlice) registry.lookup("IAlice");

```

Listing 4.3: Method call for looking up an **IAlice** implementation

The process of looking up Servlets in the registry is illustrated in the sequence diagram in Figure 4.6. When an object performs a lookup at the local registry, this request is forwarded to the remote registry Servlet, which returns the previously stored entry data shown in Listing 4.2. The local registry then uses this data to construct a stub object, which it returns to the object performing the lookup request.

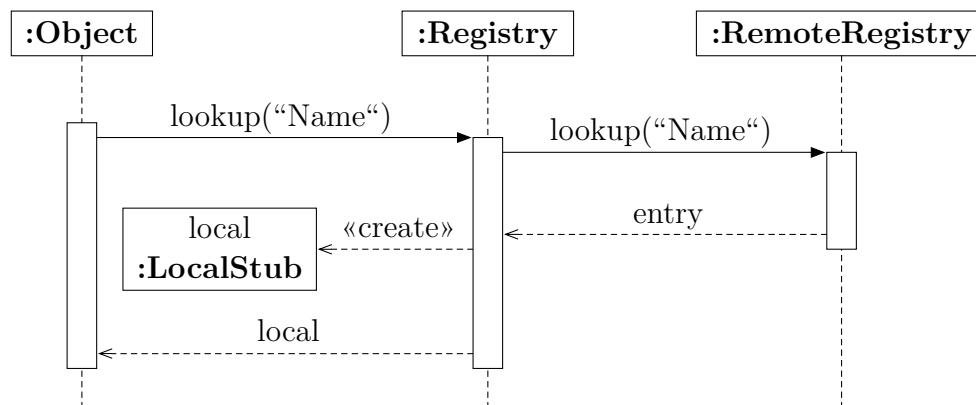


Figure 4.6: Sequence diagram showing a registry lookup

4.2.4 Communication Protocol

The actual RPC functionality for the Servlets, i.e., the communication protocol, is implemented in the stubs returned from the registry lookups. These stubs intercept the method calls performed on them, as illustrated in Figure 4.7. When a stub intercepts a method call, it serializes all relevant information, i.e., method name, parameters, and arguments, to a JSON string and forwards it to the remote destination using a POST request.

For example, a serialized call of a component class's **setContext** method could look like the JSON string shown in Listing 4.4. Besides the method name and an array of serialized argument objects, it contains two arrays specifying the method's parameters.

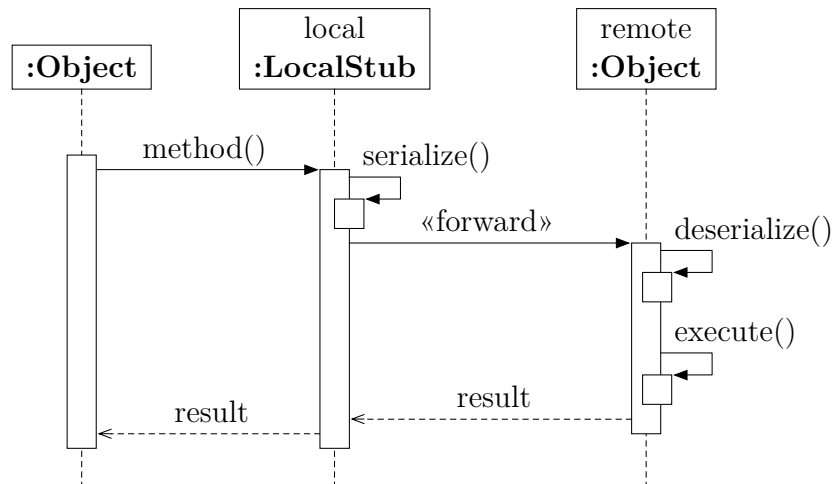


Figure 4.7: Sequence diagram showing a method call on a stub

The *formalParameters* array consists of the type names of the formal parameters and is used to find the correct method to invoke at the destination, whereas the *actualParameters* array specifies the actual type names of the serialized arguments. These type names are required during deserialization in order to recreate the serialized objects. For example, the argument in Listing 4.4 is deserialized as **AliceContext** object and passed to the **setContext** method which expects an **Object** argument.

```

1 {
2   "name": "setContext",
3   "formalParameters": ["java.lang.Object"],
4   "actualParameters": ["repository.alice.AliceContext"],
5   "arguments": [{"bobPort": "IBob"}]
6 }

```


Listing 4.4: JSON representation of a method call

When the data is received at the destination, the remote object deserializes the JSON string and executes the appropriate method with the specified arguments. Finally, it returns the result of the method call via the HTTP response of the POST request. This return value is then forwarded from the stub to the invoking object, resulting in a transparent and synchronous method call.

4.2.5 Web User Interface

The web user interface provides functions similar to the CLI user interface of Java SE ProtoCom. It is used to specify the URL of the registry and to start the different modules of the performance prototype, i.e., system and resource containers. Furthermore, it displays a log with all messages coming from locally started modules.

Figure 4.8 shows a sketch of this user interface, which is implemented as a Servlet and uses HTML and JavaScript to display its elements.

 **ProtoCom** Performance Prototype

Registry Location

Modules

Container: Workstation	Started	Stop
System: AliceBobSystem	Stopped	Start

Local Log

12:30:05 INFO Start container 'Workstation'
12:42:16 INFO Invoke 'IAlice.callBob'

Figure 4.8: Sketch of the HTML user interface

4.2.6 Usage Scenarios

Whereas most of the interaction with the performance prototype is managed through the web user interface, performance engineers use external HTTP load generators to execute usage scenarios. As discussed in Section 3.3.2, the transformation of usage scenarios is not part of this thesis. Instead, this section presents a proof of concept for external usage scenarios with Apache JMeter.

The simulated users of a usage scenario always interact with the system. Hence, they perform operations according to the system's provided roles. Because the system is accessed over its port Servlet, performance engineers can use JMeter to generate POST requests for these operations according to the communication protocol explained in Section 4.2.4.

JMeter provides a variety of so-called logic controllers that allow the modeling of complex test plans with control flow similar to usage scenarios, including loops and conditionals. Therefore, usage scenarios of the PCM can generally be expressed using these test plans. Table 4.2 and Table 4.3 show the JMeter settings of a closed workload example scenario with a population of fifteen users and a think time of two seconds, where the simulated users repeatedly call the `callBob` method of

the *AliceBobSystem*. The test plan for this scenario consists of three entities: a *Thread Group* that defines the number of threads (i.e., users) and their request looping behavior, a *Constant Timer* that adds a delay (i.e., think time) between two consecutive requests in a thread, and an *HTTP Request* that specifies the system's location and the request method.

Table 4.2: Population and think time settings

<i>Thread Group</i>		<i>Constant Timer</i>	
Setting	Value	Setting	Value
Threads	15	Thread Delay	2000
Loop Count	Forever		

Table 4.3: HTTP request settings

<i>HTTP Request</i>	
Setting	Value
Location	localhost
Port	8080
Path	Prototype/IAliceBobSystem
Method	POST

The body of the POST request for the method call is given in Listing 4.5. Since this reference implementation does not specify any parameters for the methods of the **IBob** interface, all three parameter-related arrays are empty.

```

1 {
2   "name": "callBob",
3   "formalTypes": [],
4   "actualTypes": [],
5   "arguments": []
6 }
```

Listing 4.5: Body of the method call's POST request

4.3 Transformations

According to AC-MDSD, the code of a reference implementation can generally be divided into three categories: individual code, schematic repetitive code, and generic code. However, performance prototypes do not contain individual code, because their business logic is defined through Service Effect Specifications (SEFFs) in the model. ProtoCom uses these SEFFs to generate artificial resource demands

by producing appropriate control and data flow as well as calls to framework functions in the transformed methods.

Following the discussion of generic code (i.e., framework functionality) in Section 4.2, this section explains the transformations derived from the schematic repetitive code of the reference implementation.

Table 4.4 lists the files that are generated for each PCM entity during the transformations, along with the classes responsible for providing their data (content providers). After the description of Java code generation in Section 4.3.1, Section 4.3.2 explains some of the transformation classes for *BasicComponent* and *System* entities.

The transformations of *Allocation* and *ResourceEnvironment* entities for the Java EE Servlet target platform are similar to the transformations for Java SE and are, therefore, not further discussed.

Table 4.4: Generated Java files and content providers for each PCM entity type

BasicComponent
Component class (ServletBasicComponentClass)
Component interface (ServletBasicComponentInterface)
Port classes (ServletBasicComponentPortClass)
Provided role interfaces (ServletOperationInterface)
Context class (ServletBasicComponentContextClass)
Context interface (ServletBasicComponentContextInterface)

System
System class (ServletSystemClass)
System interface (ServletComposedStructureInterface)
Port classes (ServletComposedStructurePortClass)
Prototype entry point (ServletSystemMain)

Allocation
Allocation storage class (ServletAllocationStorage)

ResourceEnvironment
Resource environment class (ServletResourceEnvironment)

4.3.1 Code Generation

The *Language* component of ProtoCom (cf. Figure 3.1) provides classes for generating and storing files (i.a., code, text, and XML) and for managing language constructs, e.g., methods. All of these classes are independent from target technologies and PCM entities, which allows them to be reused for different transformations.

For example, most entity types are transformed to classes. Their transformations can, therefore, take advantage of the same functionality for classes in the *Language* component. Furthermore, this separation allows to provide several target technologies for a single language while ensuring reusability of language-specific classes.

The class diagram in Figure 4.9 shows an excerpt of the classes in the *Language* component that are required for generating Java classes. The **GeneratedFile** class with the generic type **IJClass** is at the top of the inheritance hierarchy and represents a single Java class file to be generated. Its subclasses **JCompilationUnit** and **JClass** specify templates that are used to generate the actual Java code. However, the contents for these templates, e.g., the methods and the name of the super class, depend on the target technology and the entity being transformed. Hence, the specification of these contents is implemented by a technology-specific content provider that is connected to the **GeneratedFile** object before the transformations start, as explained in Section 4.3.2.

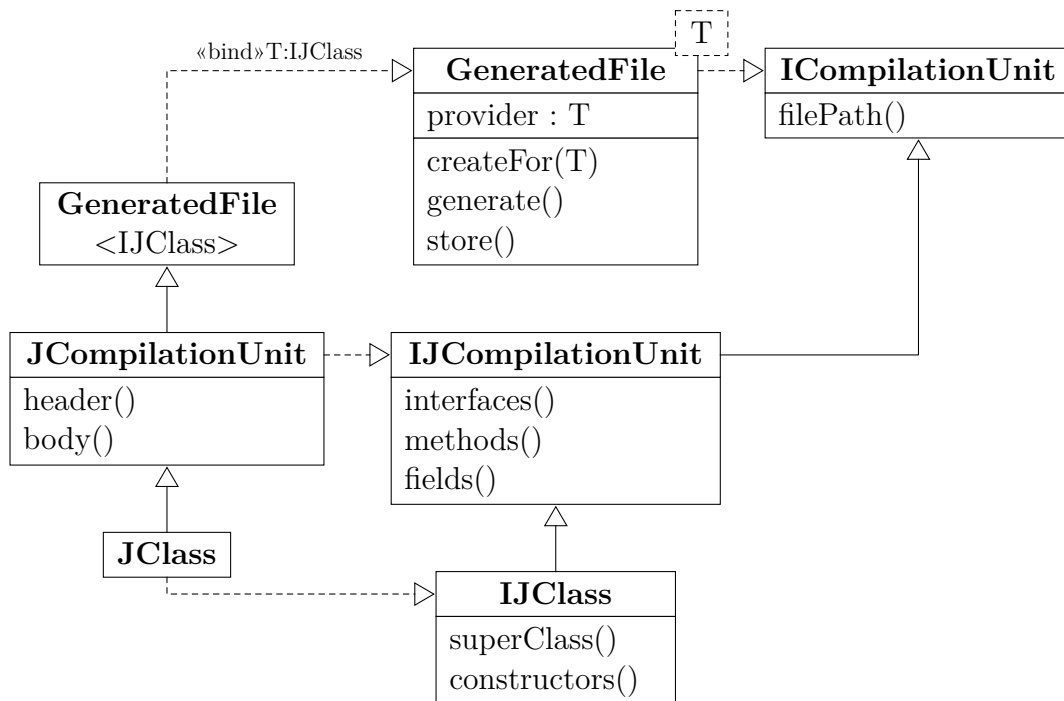


Figure 4.9: Excerpt of the classes in the *Language* component³

In addition to managing file generation, the *Language* component contains classes that content providers can use to specify methods (**JMethod**), annotations (**JAnnotation**), and fields (**JField**) in a structured way. For example, a **JMethod** object can be constructed as shown in Listing 4.6. After creating the object, several constructor method calls are chained together, resulting in a single construction expression.

³The diagram does not show all methods of the classes.

```

1 new JMethod()
2   .withVisibilityModifier("public")
3   .withName("start")
4   .withParameters("String componentId, String assemblyContext")
5   .withThrows("ModuleStartException")
6   .withImplementation(''
7     // Java code
8     '')

```

Listing 4.6: Construction of a **JMethod** object

The transformation of the created **JMethod** object is handled by the Xtend template shown in Listing 4.7. First, it iterates over the list of annotations and delegates their transformation to the corresponding **JAnnotation** object. Afterwards, the template generates the code for the modifiers, return type, signature, and thrown exceptions in lines 3 to 7. Finally, it inserts the body specified by the **withImplementation** method. If no body is specified, the generated method is treated as abstract.

```

1 '''
2 «FOR a : annotations SEPARATOR '\n'»«a.generate»«ENDFOR»
3 «visibilityModifier»
4   ⇨ «IF isStatic» «staticModifier»«ENDIF»_
5   ⇨ «returnType»_
6   ⇨ «name»(«parameters»)
7   ⇨ «IF throwsType != null» throws «throwsType»«ENDIF»
8   ⇨ «IF body != null» {
9     «body»
10  }«ELSE»;«ENDIF»
11 '''

```

Listing 4.7: Xtend code of the method transformation

Evaluating this template for the **JMethod** object in Listing 4.6 results in the Java code shown in Listing 4.8. During the evaluation, missing data for visibility and return type are generated automatically. For example, if the construction of a method does not specify a return type, the transformation defaults to **void**.

```

1 public void start(String componentId, String assemblyContext)
2   ⇨ throws ModuleStartException {
3   // Java code
4   }

```

Listing 4.8: Generated Java code of the method

4.3.2 Entity Transformations

The transformations of PCM entities are performed by connecting file generators to appropriate content providers from ProtoCom's *Technology* component (cf. Figure 3.1) like the ones listed in Table 4.4. These transformations are explained in this section by the example of *BasicComponent* entities. Subsequently, this section describes the characteristics of the *System* transformations.

BasicComponent

The class diagram in Figure 4.10 shows the classes of the *Traversal* and *Technology* components involved in transforming *BasicComponent* entities. The purpose of the classes in the *Traversal* component (i.a., **PcmRepresentative** and **XBasicComponent**) is to determine which files are generated for each entity and to connect them to the appropriate content providers. An instances of the generic **PcmRepresentative** class represents a single entity in a PCM instance—in this case a *BasicComponent*—and starts the transformation. The specification of the generated files according to Table 4.4 is handled by the **JeeServletBasicComponent** subclass.

The diagram also shows the content provider for the component class (**ServletBasicComponentClass**), which implements the **IJClass** interface, corresponding to the generic type of **GeneratedFile**, as discussed in Section 4.3.1.

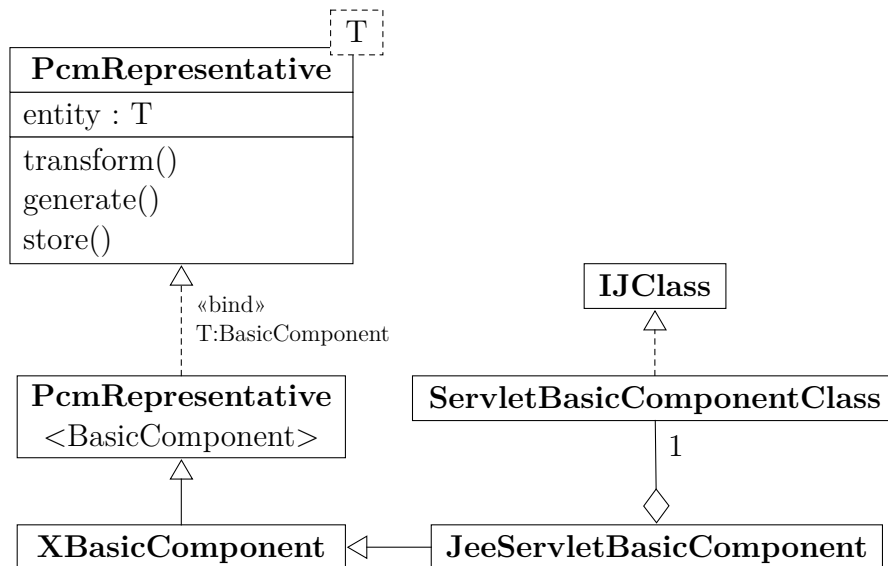


Figure 4.10: Excerpt of the classes in the *Traversal* and *Technology* components

The sequence diagram in Figure 4.11 illustrates how the connection of these classes is accomplished. When the transformation is initiated at a **JeeServletBasicComponent** object, a call to the `generate` method sets up the output files.

First, the method creates the content provider (`ServletBasicComponentClass`) for the file. Afterwards, it creates a `JClass` object and connects the content provider to it using the `createFor` method. Following the completion of the setup, a call to the `store` method starts the actual transformation for the file.

The last method call (*) represents the retrieval of data from the content provider. At this point, the `JClass` object repeatedly queries the content provider for information according to the `IJClass` interface, e.g., class name, file path, and methods implementing SEFFs. The content provider collects this information from the PCM entity, processes it if necessary, and returns it to the `JClass` object.

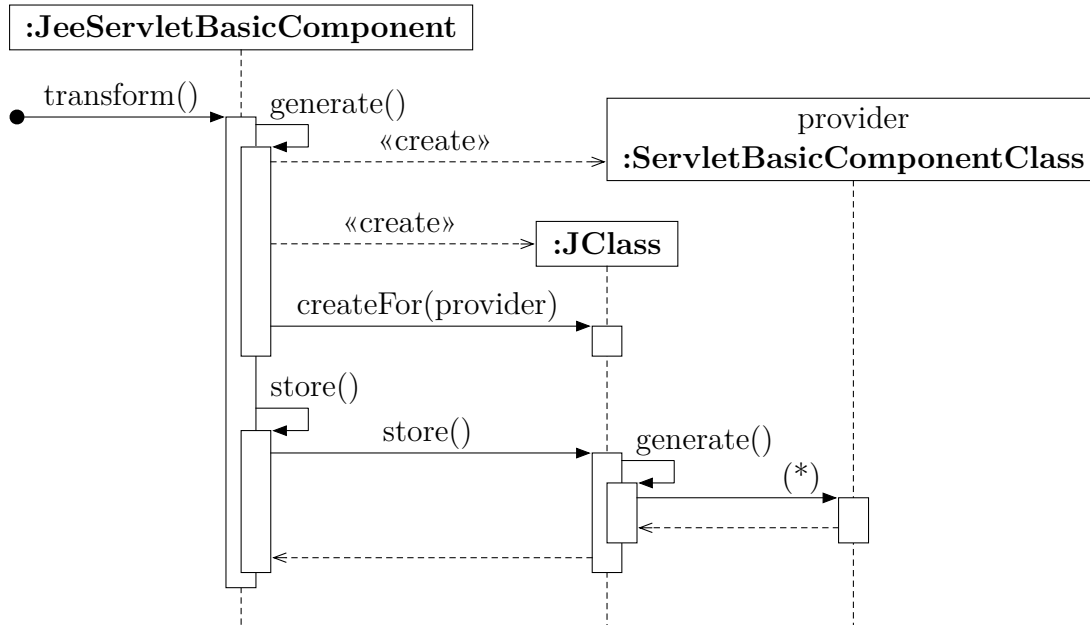


Figure 4.11: Sequence diagram showing the transformation process

For simplicity, this description of the file generation only considers the main class of the transformed *BasicComponent* entity. However, the generation of the files for interfaces, port classes, and the context class follows the same concepts. Their setup is implemented in the `generate` method of the `JeeServletBasicComponent` class shown in Listing 4.9. In addition to the objects for the component's main class file, it creates pairs of `JClass`/`JInterface` objects and according content providers for all other files. For the port classes, it iterates over the list of the component's provided roles and generates a file for each of these roles.

```

1 class JeeServletBasicComponent extends XBasicComponent {
2   override protected generate() {
3
4     // Class for the component
5     generatedFiles.add(
6       injector.getInstance(typeof(JClass))
  
```



```

7      .createFor(new ServletBasicComponentClass(entity))
8    )
9
10   // Interface for the component
11   generatedFiles.add(
12     injector.getInstance(typeof(JInterface))
13     .createFor(new ServletComponentClassInterface(entity))
14   )
15
16   // Classes for the ports
17   entity.providedRoles_InterfaceProvidingEntity.forEach[
18     generatedFiles.add(
19       injector.getInstance(typeof(JClass))
20       .createFor(new ServletBasicComponentPortClass(it))
21     )
22   ]
23
24   [...]
25 }
26 }

```

Listing 4.9: Xtend code of the `JeeServletBasicComponent` class

System

The transformation of *System* entities is similar to the transformation of *BasicComponent* entities. However, since only one *System* entity exists in a PCM instance, it can—from the perspective of performance engineers—be regarded as the entry point of the performance prototype. Therefore, the system transformation also performs tasks that have to be performed once in the whole prototype transformation process. For example, it copies static files and generates metadata. This includes any settings (e.g., for the classpath and runtime environment), external libraries (e.g., for JSON serialization), and HTML, CSS, and JavaScript for the web user interface.

Furthermore, the system transformation generates a Servlet class using the `ServletSystemMain` content provider that presents the web user interface for performance engineers described in Section 4.2.5.

5 Interpretation Phase

This chapter finalizes the analysis of the ProtoCom transformations by interpreting the measurements taken during the collection phase. First, Section 5.1 answers the question specified in the GQM plan on the basis of these measurements. Afterwards, Section 5.2 evaluates why the goal of this thesis is accomplished to a certain extent, followed by a discussion of possible threats to the validity of the obtained results in Section 5.3.

5.1 Answers to Questions

The GQM plan for this thesis specifies two questions regarding the extensibility of ProtoCom: “*How extensible are the ProtoCom transformations?*” and “*How extensible is the ProtoCom framework?*”. However, the evaluation of the analysis in this section focuses mainly on the transformations and not on the framework because it became apparent early in the collection phase that little functionality of the existing framework could be reused. Hence, most of the required functionality had to be reimplemented for the Java EE Servlet technology. Details on the results of this implementation are discussed at the end of this section.

Regarding the transformations, ProtoCom contains two parts that were reused for the extensions: the *Language* component providing Java code and file generation and the framework of the *Traversal* component, which provides the base functionality for transforming PCM entities.

5.1.1 Language Component

First, measurements were taken for the *Language* component according to the metrics specified in Section 3.4.3, i.e., Non-Commenting Source Statements (NCSS), concrete classes (CC), abstract classes (AC), abstractness (A), afferent couplings (Ca), efferent couplings (Ce), and instability (I). These measurements, grouped by Java packages, are listed in Table 5.1. Each row in this table lists the results on the top level of the particular package. Hence, the results for the *lang* package refer only to the package itself, not to its subpackages indicated by the gray arrows.

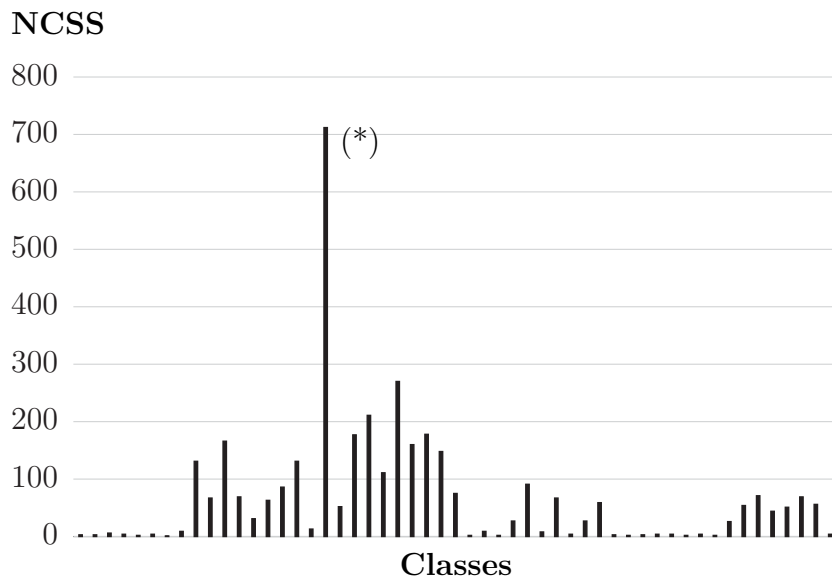
According to the NCSS measurements, the Java code generation makes up the largest part of the component. The remaining packages contain classes responsible for creating other file types, e.g., XML for classpaths and settings.

Table 5.1: Measurements for the *Language* component

Package	NCSS	CC	AC	A	Ca	Ce	I
lang	57	0	2	1.00	5	7	0.58
↪ java	3143	15	11	0.42	23	17	0.43
↪ manifest	150	2	3	0.60	5	2	0.29
↪ prefs	81	1	1	0.50	2	1	0.33
↪ properties	37	1	1	0.50	3	1	0.25
↪ txt	76	1	1	0.50	2	2	0.50
↪ xml	458	8	7	0.47	6	2	0.25
Total	4002	28	26	0.48	46	32	0.41

The fact that the abstractness lies around 0.50 for most packages and amounts to 0.48 in total indicates good extensibility of the component because it provides a corresponding interface or abstract class for almost each concrete class, thereby decoupling the specification of its functionality from the actual implementation.

Additionally to the measurements on the package level, the analysis considers the distribution of NCSS for all classes in the *Language* component in order to discover possible uses of the god object anti-pattern [Rie96], as illustrated in Figure 5.1.

Figure 5.1: NCSS distribution in the *Language* component

Most of the classes are comparatively small and contain at most 300 NCSS. However, the **JavaNames** class (*) stands out with more than 700 NCSS because it is used to translate the names of all PCM entities to valid Java names. Therefore, each entity transforming class depends on the **JavaNames** class. This dependency negatively affects the extensibility of the component because additional target

technologies may require changes in the name translations that can affect existing transformations.

The coupling and instability measurements, on the other hand, were incapable of providing further insights regarding the current extensibility. For example, most of the efferent couplings of the *java* package are attributable to PCM entity classes used by the **JavaNames** class. Hence, the number of efferent couplings will increase only insignificantly with the number of target technologies added in the future. Instead, the instability will decrease when more packages of these additional technologies use the *Language* component due to additional afferent couplings.

Concerning the modifications of the *Language* component during the extensions, the hypotheses in the GQM plan state that at most minor changes were expected. However, it turned out that this component lacked code generation for some Java features. For example, the extensions required class- and method-level annotations, which were unable to be generated with the existing capabilities. Furthermore, the component lacked the generation of settings files for Eclipse projects. Overall, 6.52% of the code in the modified *Language* component account for functionality added during the implementation of the extensions, as shown in Table 5.2. In conclusion, the required effort for these additions was relatively low but still higher than expected.

Table 5.2: Added classes and NCSS in the *Language* component

Package	Classes		NCSS		
	Before	After	Before	After	Delta
lang	2	3	57	100	75.44%
↪ java	26	28	3143	3325	5.79%
↪ manifest	5	5	150	150	0.00%
↪ prefs	2	2	81	81	0.00%
↪ properties	2	2	37	37	0.00%
↪ txt	2	2	76	76	0.00%
↪ xml	15	19	458	494	7.86%
Total	77	84	4002	4263	6.52%

Although the requirements for the extensions in this thesis were implemented, there are still Java features that cannot be generated currently. For example, the generation of anonymous classes and enumerations is missing but may be useful for target technologies added in the future.

Finally, Table 5.3 lists the amount of documented classes and methods in each package of the *Language* component. The analysis shows that only 33.33% of the classes and 22.33% of the methods are documented. From the perspective of transformation engineers, this lack of documentation interferes with the component's extensibility because it results in increased implementation time for the extensions, as discussed in Section 5.1.3.

Table 5.3: Documentation of classes and methods in the *Language* component

Package	Clas.	Doc.	Rate	Met.	Doc.	Rate
lang	2	2	100.00%	5	4	80.00%
↪ java	26	16	61.54%	45	19	42.22%
↪ manifest	5	0	0.00%	10	0	0.00%
↪ prefs	2	0	0.00%	7	0	0.00%
↪ properties	2	0	0.00%	3	0	0.00%
↪ txt	2	0	0.00%	2	0	0.00%
↪ xml	15	0	0.00%	31	0	0.00%
Total	54	18	33.33%	103	23	22.33%

5.1.2 Traversal Framework

As for the *Language* component, the measurements for the framework of the *Traversal* component were taken according to the metrics specified in Section 3.4.3 and grouped by Java packages. Compared to the measurements for the *Language* component, the measurements given in Table 5.4 allow only limited statements about the component’s extensibility. For example, the low abstractness of 0.08% indicates a high coupling of the classes in the component. However, all classes are meant to be extended by the technology-specific implementations. Hence, they are used like abstract classes, despite not being declared as abstract.

Table 5.4: Measurements for the framework of the *Traversal* component

Package	NCSS	CC	AC	A	Ca	Ce	I
traversal.framework	30	0	1	1.00	8	9	0.53
↪ allocation	4	1	0	0.00	2	1	0.33
↪ repository	151	8	0	0.00	6	4	0.40
↪ resourceenv.	4	1	0	0.00	2	1	0.33
↪ system	3	1	0	0.00	6	1	0.14
↪ usagescenario	4	1	0	0.00	2	1	0.33
Total	196	12	1	0.08	26	17	0.40

Similarly, the small set of currently supported target technologies results in a low amount of afferent couplings, which, in turn, leads to a comparatively high total instability of 0.40. However, the framework of the *Traversal* component is designed for being reused by the transformations of each target technology. Thus, the instability will decrease with each target technology added in the future due to additional afferent couplings.

Contrary to the measurements, this part of the *Traversal* component supports the extensibility of the ProtoCom transformations because it is sufficiently documented (i.e., the functionality of 13 out of 14 classes is described with Javadoc) and provides a small layer of abstract functionality that is used by the transfor-

mations of all target technologies. Furthermore, no additions to this part of the *Traversal* component were required during the implementation of the extensions. Hence, no further measurements are presented here.

5.1.3 Implementation

Regarding the implementation of the Servlet target technology in the *Technology* component and the framework, NCSS and implementation time in hours were measured during the analysis, as shown in Table 5.5.

Table 5.5: NCSS and implementation time

Transformations			Framework		
Package	NCSS	Hours	Feature	NCSS	Hours
tech.servlet	296	4.75	Framework	238	14.00
↪ allocation	84	1.00	↪ Registry	241	12.25
↪ repository	955	19.50	↪ Comm.	115	15.25
↪ resourceenv.	33	0.25	↪ Web UI	157	8.00
↪ system	341	10.25			
↪ util	191	2.00			
Total	1900	37.75	Total	751	49.50

Transformations

The implementations of the transformations in the *repository* package and the *system* package were the most time-consuming and amount to the largest part of the NCSS.

However, such correlations are nonexistent for some of the other packages. For example, the *util* package contains 191 NCSS but required an implementation time of only 2.00 hours. This deviation is caused by the fact that many parts of the transformations could be copied from the existing Java SE transformations. This copying of functionality indicates bad reusability because common features should be made available through a single component that is accessible to the transformations for every target technology. Similar observations hold for the *tech.servlet* package, which provides base functionalities for its subpackages.

Furthermore, the lack of documentation as described in Section 5.1.1 harms the comprehensibility of the reused features from the *Language* component and, thus, contributed to the implementation time.

Framework

For the most part, the reimplementing of the framework was required due to its strong orientation towards Java SE. However, there were also unexpected issues that considerably increased the implementation time. For example, the resource

strategies used for generating resource demands could have been reused but proved to be incompatible with the SAP HANA Cloud, due to external dependencies and version conflicts with libraries provided by the platform. Therefore, the framework implemented in this thesis provides only the fundamental features for running generated Java EE Servlet performance prototypes, i.e., a user interface and inter-component communication. Reimplementing the resource strategies or making them compatible with the SAP HANA Cloud remains as future work.

Conclusion

In summary, the hypotheses for both questions of the GQM plan turned out to be mostly correct. Despite some weaknesses, the transformations have overall good extensibility characteristics, whereas the framework lacks extensibility measures in its design.

5.2 Goal Attainment

According to the GQM plan, the goal of this thesis (cf. Section 3.1) is to “*analyze the architecture of ProtoCom for the purpose of maintainability assessment with respect to extensibility from the viewpoint of transformation engineers in the context of performance prototype transformations*”.

Section 5.1 successfully answers the two questions derived from this goal. However, a detailed extensibility analysis was possible only for the transformations. Although the framework required to be reimplemented to a large extent for the selected extension scenario, it may have sufficient extensibility characteristics for smaller changes and additions, e.g., encrypted communication between components, as discussed in Section 3.3.1. Furthermore, a complete analysis of ProtoCom requires additional target languages to be implemented (cf. Section 3.3.1).

Thus, the results of the analysis provide a starting point for future investigations and enhancement to both the transformations and the framework.

5.3 Threats to Validity

The measurements taken during the implementation can be treated as results of an experiment, which raises questions about their validity. For the purpose of answering these questions, Wohlin et al. [WRH⁺12, Section 8.7] specify an extensive set of possible threats to the validity of experimentation results. However, only few of these threats are applicable for this thesis. Therefore, this section uses a simplified approach that discusses only the most important threats and states corresponding counter-measures taken before and during the measurements.

Low Statistical Power The extension scenario selected for the implementation covers only a single target technology. Hence, the measurements have a low sta-

tistical power. The results regarding the extensibility may be partially caused by the choice of the extension scenario and could be impossible to transfer to other scenarios. In order to verify the results of this thesis, further extension scenarios have to be implemented and evaluated.

Reliability of Measures When performing measurements twice for the same object under consideration, the outcome should be identical. For the code-related metrics described in Section 3.4.3, this is expected to hold true because they are precisely specified. One threat to their reliability is that the measurements were performed using software tools, namely JavaNCSS and JDepend, which may contain bugs that compromise the results. However, manual measurements were performed on a sample basis in order to verify the output of these tools.

The measurements of the implementation time, on the other hand, are unlikely to be reproducible because they are highly dependent on the skills, discipline, and determination of the transformation engineer conducting the implementation. Furthermore, the learning effect during the implementation interferes with the reproducibility (cf. maturation threat). An according controlled experiment with a large number of participants can conquer this threat.

Regarding the NCSS metric, there is a threat that the measurements are unable to reflect the actual amount of code added during the implementation because these measurements are based on the output of the Xtend compiler. Although the Java code generated by this compiler is comprehensible and resembles the original Xtend code, it may still contain a considerable amount of additional source statements, e.g., when using templates.

Maturation The measurements for the implementation time metric were performed over a period of about two and a half weeks. Two threats arise when taking measurements over such a time frame. First, the motivation of the implementing transformation engineer can vary due to several influences. For example, the discovery of unexpected bugs can decrease the motivation. Second, a learning effect can be achieved during the measurements. This holds especially true for the transformations presented in this thesis because most of them work in a similar fashion. Therefore, transformations implemented later during development were completed more quickly. Controlled experiments should be conducted for investigating these issues in more detail.

Random Irrelevancies in Experimental Setting There is a risk that the implementation process is interrupted or impaired by unexpected events, e.g., incoming phone calls. To overcome this threat, the implementation time was measured using the software Tyme, which allowed to manage time tracking with a low overhead.

Fishing The threat of looking for a certain outcome of the measurements (“fishing”) arises when the metrics are selected depending on the desired outcome. In this thesis, the metrics were selected before implementing the extension scenario, thus ensuring an interpretation that is independent of prior knowledge about the object under consideration.

6 Related Work

This chapter presents a short overview of related work regarding the quality of model transformations and performance prototyping approaches.

Quality of Model Transformations In his PhD thesis, van Amstel [vA11] assesses the quality of model transformations. However, he mainly compares the model transformation quality of different transformation languages, e.g., ATL, QVT-O, and Xtend. Furthermore, a new version of Xtend (Xtend 2, cf. Section 2.2) was released since the publication of his thesis. This new version provides a different feature set than the version examined by van Amstel. Hence, the results are only indirectly applicable for this thesis.

Furthermore, Lehrig [Leh12] presents Java code metrics for assessing the quality of model transformations in his master’s thesis. However, he focuses on model-to-model transformation and ignores model-to-text transformations in the context of performance prototyping.

Performance Prototyping To the best of the author’s knowledge, performance prototyping is currently only supported by ProtoCom. In his PhD thesis, Becker [Bec08] provides the groundwork for ProtoCom and for transforming PCM instances to performance prototypes using Java EE EJBs (Enterprise Java Beans). However, the performance prototypes generated by these transformations suffer from several usability issues. For example, the generated code requires manual adjustments after the transformations. Furthermore, the deployment process of the generated performance prototypes proved to be inefficient.

Building on this work, Lehrig and Zolynski [LZ11] present an improved version of ProtoCom that aims to resolve these shortcomings. They extend the transformations and the framework of ProtoCom such that the generated performance prototypes target the Java SE technology and use Java RMI for inter-process communication. However, the transformations in this version of ProtoCom proved to be slow and inextensible due to the use of Xpand¹ templates as a means for model-to-text transformations. Therefore, Lehrig and Zolynski provide a reimplemented version of ProtoCom (“ProtoCom 3”) [KIT13] that replaces the Xpand templates with templates for the programming language Xtend 2 (cf. Section 2.2).

Since the release of Becker’s PhD thesis, a new version of Java EE EJBs was published, providing features that simplify the transformations. Furthermore, there is a need for performance prototype transformations targeting multiple platforms

¹<http://www.eclipse.org/modeling/m2t/?project=xpand>

[LLK13]. These developments led to the addition of new Java EE EJB transformations by Giacinto and Lehrig [GL13]. However, these transformations are provided only on a conceptual level and also lack an implementation.

All of the approaches in the category of performance prototyping have in common that they lack an investigation of their extensibility.

7 Conclusions

This chapter concludes this thesis. First, Section 7.1 gives a summary of the presented work including the used methodology and the analysis results. Afterwards, Section 7.2 outlines the benefits for transformation and performance engineers. Finally, Section 7.3 points to future work regarding the Java EE Servlet transformations and ProtoCom in general.

7.1 Summary

In order to cope with the lack of extensibility analyses of performance prototyping approaches such as ProtoCom, this thesis implements an extension for the ProtoCom transformations and their associated framework. During the extension process, it takes measurements for extensibility metrics.

For the purpose of structuring the analysis, this thesis applies the Goal Question Metric (GQM) method and, therefore, specifies a set of metrics as well as a set of hypotheses that state the expected outcome of the measurements. During the planning of the implementation, the Architecture-Level Modifiability Analysis (ALMA) method is used to examine the architecture of ProtoCom in order to find extension scenarios that are suitable for the extensibility analysis. This procedure exposes several extension scenarios and argues that an extension for an additional target technology provides the best compromise between expected implementation effort, reusability of existing components, and significance in terms of extensibility. Thus, transformations for the Java EE Servlet technology targeting the SAP HANA Cloud platform are selected for the implementation.

The construction of these transformations follows the concepts of Architecture-Centric Model-Driven Software Development (AC-MDSD). Hence, this thesis presents a reference implementation consisting of a PCM instance and its manual transformation for the Java EE Servlet target technology. This reference implementation is then used to extract the functionality for the framework (e.g., an HTTP-based RPC protocol and an HTML user interface) and to derive automated transformations for arbitrary PCM instances that are integrated into ProtoCom.

Finally, this thesis presents the quantitative data collected during the implementation and interprets them in terms of the extensibility of ProtoCom. This analysis reveals that the transformations provide overall good extensibility, whereas the framework requires significant improvements in the future.

7.2 Knowledge Gained

The results of this thesis help transformation engineers in two ways. First, this thesis provides metrics and procedures for analyzing performance prototype transformations. Transformation engineers can use these procedures for further evaluations of ProtoCom or they can apply them to evaluations of other transformations. Second, transformation engineers can use the presented results to address the extensibility issues in ProtoCom, thereby facilitating the addition of other target technologies to ProtoCom in the future.

Furthermore, the results have practical relevance for performance engineers because this thesis provides support for an additional target technology for ProtoCom, namely Java EE Servlets. This target technology allows performance engineers to evaluate the performance of software architectures on several cloud platforms. However, these transformations require further additions. Corresponding suggestions are given during the discussion of future work in Section 7.3.

7.3 Future Work

This thesis provides starting points for future work in three areas: validating the presented work, completing the transformations for Java EE Servlets, and improving the extensibility of ProtoCom for future extensions.

Validation As discussed in Section 5.3, there exist several threats to the validity of the analysis results. Most importantly, the low statistical power of the measurements should be addressed through further extensions and evaluations, e.g., for other target technologies. Furthermore, countering the maturation threat by employing different transformation engineers implementing these extensions can improve the validity of the results. Additionally, the use of other metrics may reveal extensibility characteristics that remain undiscovered in this thesis.

Completion of Transformations Regarding the transformations for Java EE Servlets, there are several PCM entities and concepts that still need to be implemented. For example, the transformations of *CompositeComponent* and *InfrastructureInterface* entities are omitted in this thesis. Furthermore, the transformation of usage scenarios to Apache JMeter test plans is discussed only on a conceptual level and based on a proof-of-concept implementation (cf. Section 4.2.6). Future work can also reduce the restrictions for the PCM input instances imposed by the current transformations. For example, the transformed systems currently lack the ability to instantiate multiple instances of the same component, as described in Section 4.1.1. Implementing this improvement will require changes in the framework and the transformation of the system.

Improvement of Extensibility Finally, there are various possibilities for future work to improve the extensibility of ProtoCom. Concerning the transformations, large amounts of existing code can be copied with only minor adjustments when adding new target technologies. Hence, the extensibility can be improved by transferring commonly used functionality to separate, reusable components, e.g., for transforming SEFFs. Furthermore, the documentation of reusable components requires enhancements in both quality and quantity. Regarding the framework, introducing more layers of abstraction will allow for easier additions of functionality. For example, increased abstraction could enable the implementation of multiple options for the same task (e.g., a command-line user interface and a web-based user interface) without making changes in the transformations.

A Deliverables

This appendix lists the deliverables for this thesis, which can be found on the accompanying DVD. The included Eclipse installation runs on Mac OS X 10.9 Mavericks with a Java 7 runtime. However, all included projects were also tested on Windows 8. The DVD contains the following items:

- This thesis in PDF format
- Brainstorming Guide in PDF format used by the participants of the brainstorming during the Architecture-Level Modifiability Analysis (ALMA)
- Eclipse Luna Modeling Tools with the following installed and configured plugins: Palladio Nightly, Web Tools Platform, SAP HANA Cloud Platform Tools, JDepend
- SAP HANA Cloud Platform SDK and SAP JVM
- Tomcat 7.0
- Eclipse workspaces with the reference implementation model project and the ProtoCom projects implementing the Java EE Servlet transformations
- Apache JMeter
- User Guide in PDF format that explains how to start and use the projects
- JavaNCSS command-line tool, which was used for measuring Non-Commenting Source Statements

Bibliography

- [FZI13] FZI Forschungszentrum Informatik am Karlsruher Institut für Technologie. Model Solvers: Palladio Software Architecture Simulator. http://www.palladio-simulator.com/science/palladio_component_model/model_solvers/. Retrieved: 07/12/2014.
- [BCR94] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The Goal Question Metric Approach. In *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc., 1994.
- [Bec08] Steffen Becker. *Coupled Model Transformations for QoS Enabled Component-Based Software Design*, volume 1 of *Karlsruhe Series on Software Quality*. Universitätsverlag Karlsruhe, January 2008.
- [BLBvV04] PerOlof Bengtsson, Nico H. Lassing, Jan Bosch, and Hans van Vliet. Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software*, 69(1-2):129–147, 2004.
- [FCH⁺08] Donald G. Firesmith, Peter Capell, Charles B. Hammons, DeWitt Lattimer, Tom Merendino, and Dietrich Falkenthal. *The Method Framework for Engineering System Architectures*. Auerbach Publications, November 2008.
- [GFA09] Tovi Grossman, George W. Fitzmaurice, and Ramtin Attar. A Survey of Software Learnability: Metrics, Methodologies and Guidelines. In *CHI*, pages 649–658, 2009.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [GL13] Daria Giacinto and Sebastian Lehrig. Towards Integrating Java EE into ProtoCom. In *KPDAYS*, pages 69–78, 2013.
- [HKR11] J. Happe, H. Koziolk, and R. Reussner. Facilitating Performance Predictions Using Software Components. *Software, IEEE*, 28(3):27–33, May 2011.
- [KIT13] Karlsruhe Institute of Technology. ProtoCom - SDQ Wiki. <http://sdqweb.ipd.kit.edu/wiki/ProtoCom>. Retrieved: 07/12/2014.

- [KKC00] Rick Kazman, Mark Klein, and Paul Clements. ATAM: Method for Architecture Evaluation. Technical Report CMU/SEI-2000-TR-004, Software Engineering Institute, Carnegie Mellon University, 2000.
- [Leh12] Sebastian Lehrig. Assessing the Quality of Model-to-Model Transformations Based on Scenarios. Master thesis, Software Engineering Group, University of Paderborn, Software Engineering Group, Paderborn, Germany, October 2012.
- [LLK13] Michael Langhammer, Sebastian Lehrig, and Max E. Kramer. Reuse and Configuration for Code Generating Architectural Refinement Transformations. In *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, VAO '13, pages 6:1–6:5, New York, NY, USA, 2013. ACM.
- [LTM⁺11] Fang Liu, Jin Tong, Jian Mao, Robert Bohn, John Messina, Lee Badger, and Dawn Leaf. NIST Cloud Computing Reference Architecture (Special Publication 500-292), September 2011.
- [LZ11] Sebastian Lehrig and Thomas Zolynski. Performance Prototyping with ProtoCom in a Virtualised Environment: A Case Study, November 2011.
- [Mar03] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.
- [MG11] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing (Special Publication 800-145), September 2011.
- [Nah03] Fiona Fui-Hoon Nah. A Study on Tolerable Waiting Time: How Long Are Web Users Willing to Wait? In *AMCIS*, page 285, 2003.
- [RBB⁺11] Ralf Reussner, Steffen Becker, Erik Burger, Jens Happe, Michael Hauck, Anne Koziolk, Heiko Koziolk, Klaus Krogmann, and Michael Kuperberg. The Palladio Component Model. Technical report, KIT, Fakultät für Informatik, Karlsruhe, 2011.
- [Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996.
- [SVC06] Thomas Stahl, Markus Völter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, May 2006.
- [vA11] Marinus Franciscus van Amstel. *Assessing and Improving the Quality of Model Transformations*. PhD thesis, PhD thesis, Eindhoven University of Technology, 2011.

- [vSB99] Rini van Solingen and Egon Berghout. *The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development*. McGraw-Hill, 1999.
- [WRH⁺12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer, 2012.