Concern-Specific Modelling Languages

An in-depth study from conceptual framework, to proof of concept, to empirical assessment.

Maximilian Schiedermeier
Doctor of Philosophy

School of Computer Science
McGill University
Montréal, Québec, Canada

April 4, 2024

A thesis submitted to McGill University in partial
fulfillment of the requirements of the degree of
Doctor of Philosophy

Abstract

Modern software needs to cope with the ever-increasing complexity of systems, and hence, reducing complexity is a primary objective of software engineering. Notably for engineering tasks that showcase high bridging of abstractions, Domain-Specific Languages (DSL) bear the potential to effectively amend existing corresponding toolchain techniques, by further assisting software engineers in this matter with reduced accidental complexity.
Unfortunately, standard toolchain techniques, which are commonly set on Model-Driven Engineering (MDE) and Aspect-Oriented Modelling (AOM) techniques, in particular Concern-Oriented Reuse (CORE), are not easily amended with DSLs. MDE applies Separation of Concerns by turning software development into a process of model production and refinement, and CORE sets on AOM techniques to improve modularization for crosscutting concerns. Both techniques usually focus on General Purpose Languages (GPLs), for additional languages hinder existing MDE processes and AOM modularization, even though DSL integration would be beneficial to mitigate accidental complexity.

The interest of this thesis is to investigate what is required to integrate tailored languages with the existing MDE/CORE techniques and assess the viability and effectiveness of such an endeavour. The starting point is the integration of custom languages, next custom transformations to ensure compatibility, into CORE concerns, which coins the term Concern-Specific Languages (CSL).

I start by investigating common DSL challenges and existing Model-Driven Engineering (MDE) techniques, to derive *FIDDLR*, a general methodological framework for the design of CSL toolchains. Afterwards, I assess in detail two sample CSL-enabled toolchains, which both target engineering tasks with inherent high bridging of abstraction, and mismatch on standard

GPL-provided concepts. Those are *RESTify*, for the exposure of existing functionality through a REST API, and *AUTHify*, to enable additional secure access delegation. Finally, I empirically assess the measurable effects of one sample toolchain on general software engineering goals "development time" and "product correctness".

My research underscores the general viability of CSL-enabled concerns, which demonstrably allows the creation of advanced toolchains. Empirical validation shows a significant benefit of such toolchains for a representative software engineering task. Additionally, I gained valuable insight on factors to increase user acceptance for such assistive technologies.

Abrégé

Les logiciels modernes doivent faire face à la complexité toujours croissante des systèmes et, par conséquent, la réduction de la complexité est un objectif primordial du génie logiciel. Notamment pour les tâches d'ingénierie s'étendent sur plusieurs niveaux d'abstraction, les langages spécifiques à un domaine (DSL) ont le potentiel de completer efficacement les techniques se reposant sur des ensembles d'outils existants, en assistant les ingénieurs logiciels dans cette tache tout en réduisant la complexité accidentelle. Malheureusement, les ensembles d'outils standards qui se reposent généralement sur les techniques de MDE et de modélisation orientée aspects (AOM) de l'ingénierie dirigée par les modèles, en particulier la réutilisation orientée vers les préoccupations (CORE), ne sont pas facilement entendus par les DSL.

MDE applique la séparation des préoccupations en transformant le développement logiciel en un processus de production et de raffinement de modèles, et CORE utilise les techniques AOM pour améliorer la modularisation des préoccupations transversales. Les deux techniques se concentrent généralement sur les langages à usage général (GPL), car des langages supplémentaires entravent les processus MDE existants et la modularisation AOM, même si leur intégration serait bénéfique pour atténuer davantage la complexité accidentelle.

L'intérêt de cette thèse est d'étudier ce qui est nécessaire pour intégrer des langages personnalisés aux techniques MDE/CORE existantes et d'évaluer la viabilité et l'efficacité d'une telle entreprise. Le point de départ est l'intégration de langages personnalisés, ensuite des transformations personnalisées pour assurer la compatibilité, dans les préoccupations CORE, qui définit le terme Concern-Specific Languages (CSL).

Je commence par étudier les défis DSL courants et les techniques d'ingénierie dirigée par les modèles (MDE) existantes, pour dériver FIDDLR, un cadre méthodologique général pour la conception de chaînes d'outils CSL.

Ensuite, j'évalue en détail deux exemples de chaînes d'outils compatibles CSL, qui ciblent toutes deux des tâches d'ingénierie avec un large spectre d'abstractions. Il s'agit de *RESTify*, pour l'exposition des fonctionnalités existantes via une API REST, et *AUTHify*, pour permettre une délégation d'accès sécurisée supplémentaire.

Enfin, j'évalue empiriquement les effets mesurables d'un ensemble d'outils sur les objectifs généraux de l'ingénierie logicielle "temps de développement" et "exactitude du produit". Mes recherches soulignent la viabilité générale de l'integration d'un CSL dans une préoccupation, qui permettent manifestement la création d'ensembles d'outils avancées. La validation empirique montre un avantage significatif d'un tel ensemble d'outils pour une tâche représentative de génie logiciel. De plus, j'ai acquis des informations précieuses sur les facteurs permettant d'accroître l'acceptation par les utilisateurs de ces technologies d'assistance.

Contribution

Significant parts of this thesis are based on published and peer-reviewed publications. I here provide a brief overview of how these contributions relate to the thesis content. Unless stated otherwise, all these publications were written by the student, Maximilian Schiedermeier as the first author. Jörg Kienzle and Bettina Kemme had a supervisory role in the writing process and are therefore listed as co-authors.

- General viability of adapting the CORE toolchain by tailored modelling languages, notably in the context of REST was first discussed in my doctoral symposium contribution: "*A concern-oriented software engineering methodology for micro-service architectures*" [Sch20]
  Although no content of the paper was directly integrated into the thesis, the article laid the conceptual foundation for parts I and II of the thesis. It described the possibility of integrating custom languages within a concern to streamline reuse and exemplified beneficial effects using the BookStore sample application, including an early graphical editor illustration.

- A second important conceptual foundation for the thesis was the exploration of multi-language support in the CORE reference implementation, TouchCORE: "*Multi-Language Support in TouchCORE*" [SLL+21]
  The paper was written in collaboration with several students: Bowen Li, Ryan Languay, Greta Freitag, Qiutan Wu, Hyacinth Ali and Ian Gauthier. Jörg Kienzle and Gunter Mussbacher had supervisory roles throughout the writing process. Once more no content of this paper is directly reflected in this thesis, but the contribution laid important cornerstones, notably for the technical extension of TouchCORE to support visual mappings between different modelling languages in a generic split-view.

- Bowen Lee's master thesis on "Concern-Oriented and Model-Driven Migration of Legacy Java Applications" was written under mentoring of Maximilian Schiedermeier and supervision of Jörg Kienzle. It explored technical implications to support various REST frameworks by a single concern. The corresponding section in Chapter 8 reflects insights on transformer reuse between concern variants and the corresponding Figure 8.1 is taken from Lee's master thesis.

- Part I and II of the thesis are based on the publication: "FIDDLR: streamlining reuse with concern-specific modelling languages" [SKK21] The paper lays out the essentials of the building blocks and CORE pipeline extension discussion of the first thesis part. Furthermore, most illustrations on the effects of the RESTify concern, and discussion in the Background section and RESTify section of part 2 are taken from this publication.

- Further insights on the nature of CSLs, and CSL concerns, as well as prospective effects for SE were first discussed in the ACM student research challenge contribution "*Pushing the boundaries of planned reuse with concern-specific modelling languages*"
[Sch22a]. Although no content of the paper was directly integrated into this thesis, the publication provided foundations for Chapter 8.

- The poster "The Horsemen of Empirical Research Apocalypse" [Sch23b] compiles various challenging aspects experienced throughout the practical conduct of the experiment described in part III. Once more no content was directly taken from this publication, but Section 13 is in parts based on poster insights.

- Part III is based on the publication "*Give me some REST: A Controlled Experiment to Study Domain-Specific Language Effects*" [SKK24]. Note that at the moment of thesis writing the paper is still under peer review.

Acknowledgements

I thank my supervisors Bettina Kemme and Jörg Kienzle for their patience, their encouraging feedback, for bringing in decades of research experience, and of course for giving me the possibility to pursue this PhD. Thank you also for your support, your pragmatism, and for introducing me to a rich research community.

Special acknowledgement goes also to Bo Wen Li, whom I had the pleasure to supervise as my undergraduate and later graduate student. Notably, his implementation of a generic split-view user interface for TouchCORE was an essential technical component to realizing my research.

Many thanks also to Olivier Barais for inspiring discussions around the OAuth2 protocol, and giving me access to code samples. Our discussions greatly nourished my understanding of this complex security protocol, down to implementation-level details.

I also thank Hyacinth Ali for his work on the TouchCORE language interface mappings, which were a technical requirement for my research.

Special thanks go also to Jessie Galasso and Gunter Mussbacher, for ongoing encouragement and bringing back faith into academia.

Especially I want to thank Martin Robillard, for his extremely helpful recommendations and research feedback. Furthermore, I thank Jin Guo, Oana Balmau and Giulia Alberini for their wide-open doors and uplifting conversations. Thank you too, to the entire SE seminar attendants, for the inspiring talks, thoughtful discussions, friendly atmosphere and lab lunches.

I thank Benoit Combemale and his research team for the accommodation in Rennes, beach discussions and life advice.

Finally, I want to thank Sebastien Mosser, Francis Bordeleau and Corinne Pulgar for their objective external opinions, heartfelt support and advice.

Besides these professional endorsements, I thank the many personal supporters, without whom this journey would not have been possible.

First and foremost I thank my family for their unconditional love and support. There are no words to express my gratitude, I simply know I can always count on you.

Special thanks go also to Mathieu for an amazing friendship from day one of the PhD program. Many thanks also to Matthias, Stéph, Finn and Noah for being fantastic neighbours and friends. The same holds for Joseph, Lizzy and Emma. Thank you also, Mona and Márton for your expertise, counsel and support.

And of course, I want to thank the many amazing people outside of McGill without whom the journey would not have been possible. Thank you to all those outstanding friends who never fail to cheer me up and recharge my batteries: Thank you Yanis, Mathieu and Adrien for the road trips and Rocket League sessions. Thank you Cat and Brit for the amazing camping trips. Thank you Alex and Fred for the board games and Canada arrival integration. Thank you Mél for the long walks on Mont-Royal and phone support. Thank you Nathalie and Richard for the shelter on Île d'Orléans and a unique motorcycle ride. Thank you Paul for the pandemic balcony beers and sophisticated walks along St.Laurent. Thank you Raquel for the ever-lasting Barbados vibes and emergency tequila. Thank you Keksli for your therapeutic purrs and cuddles, your hunger still outsmarts my engineering skills. Thank you Chloé for the amazing conversations and chalet trips. Thank you Marti and Blane for the Tatort evenings and supportive talks. Thanks to all Le Cams for their unmatched hospitality and support. Thank you Alex for the many supportive calls, beers and board game sessions. Thank you Denise for the phone support and Lyon news. Thank you Lola for the treehouse trip and phone support. Thank you Masha for the Frankfurt kickstart and contemplative letters. Thank you, Estelle for the cryptographic postcards.

Contents

List of Figures

List of Tables

Listings

1

Introduction

Modern software needs to cope with the ever increasing complexity of systems [Jam08], and hence, *reducing complexity* is a primary objective of software engineering. This goal is notably relevant in the context of engineering activities where complexity stems from simultaneously operating at multiple levels of abstraction. It is easy to imagine how an absence of proficiency with programming paradigms, architectural configurations or design patterns can result in engineering flaws that in turn delay development, or reduce software quality. In the worst case, a series of fatal design flaws can potentially even jeopardize project success.

Model-Driven Engineering (MDE) advocates the use of models when developing a system [Ken02, Sch06]. Models describe properties of the system under development at different levels of abstraction, and model transformations and code generators connect the different models across layers of abstraction, from high-level requirement models down to code. Software tools that incorporate such models and transformation techniques are also referred to as "*MDE tooling*" in this thesis. In MDE, the combined use of multiple modelling languages allows the developer to express properties of the system under development at different levels of abstraction and from different points of view, thus promoting *Separation of Concerns* (SoC) and reducing com-

plexity. Model transformations connect models across levels of abstraction, effectively *reusing architectural and design knowledge*, or *platform-specific development expertise* when generating code.

Despite MDE's established effectiveness for SoC, MDE is by itself not free of complexity-associated challenges. Two central challenges in the context of models are avoiding *accidental complexity*, and supporting *planned reuse*.

- *Accidental complexity* arises from a conceptual mismatch of targeted modelling context and available modelling language concepts. When it is impossible to concisely represent a matter with the limited language concepts at disposition, models grow overly verbose, which renders their usage and interpretation overly complex. The established answer to this challenge is to introduce a dedicated *Domain-Specific Language* (DSL)[1]. As opposed to General Purpose Modelling Languages (GPLs), like for instance the Unified Modelling Language (UML) [RJB04], DSLs aim at provisioning concepts precisely tailored to the domain in question [AK08]. Historically, MDE relies predominantly on GPLs, which is partly because DSL tool development has a high overhead. However, over the recent years, DSLs have been on the rise [TK19].

- *Planned reuse*, as opposed to opportunistic reuse, targets the creation of models with the intent for reuse from the start. The designated models can be partial, i.e. the target unit of reuse may not describe fully operational systems components but focus on selective model fragments. Compositional approaches, e.g. based on *Aspect-Oriented Modelling* (AOM) techniques, allow for declarative mapping and a combination of such models. Note that the planned creation of partial models, for ulterior reuse, is a complex activity and requires guidance [KMA+16], which is, e.g. provided by the *Concern-Oriented Reuse* (CORE) framework. Throughout the thesis, I will often illustrate concepts on the example of the reference implementation, TouchCORE [Kie23].

In principle, a combination of the above solutions as illustrated in Figure 1.1 could counter the aforementioned challenges, and simultaneously foster mutually beneficial effects. Reusable models could be expressed in the

[1]Related literature also uses the term *Domain-Specific* ***Modelling*** *Language* (DSML). There is no difference between DSLs and DSMLs, for all Domain-Specific Languages are implicitly likewise Modelling Languages.

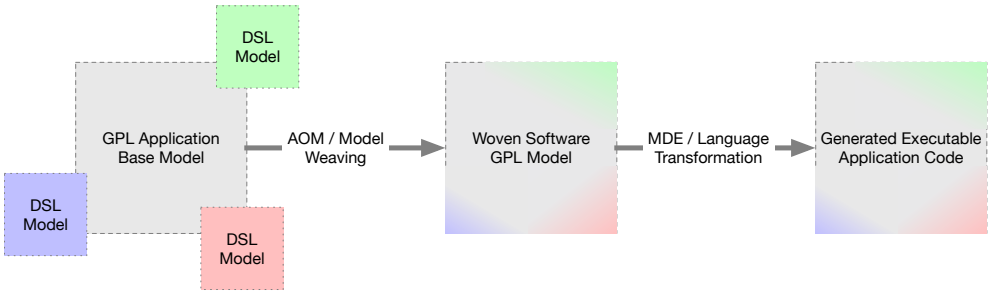

Figure 1.1: Illustration of Conceptional Technology Interplay. AOM Allows Reuse of Models by Weaving. MDE Provides Language Transformations for Code Generation

language that prescribes their intent with the least accidental complexity, transformed to GPL models using MDE model transformations, adapted to and composed with the reuse context using AOM model weaving techniques, and finally translated to executable code by MDE language transformations.

Considering these assumed benefits, one would assume that implemented adherent tooling was already a widespread technological reality. However, exactly the contrary is the case. So far, a junction of DSLs with model reuse techniques, such as CORE, has not been attempted. Presumably, this missed opportunity stems from a fundamental technical intricacy associated with any such a combination:
AOM-based model composition techniques, or weaving, is only defined within one modelling language. That is to say existing tooling like CORE allows only model reuse (and subsequent code generation), when all provided models are within the same modelling language. Naturally, existing tools focus on GPLs. In turn, this means that additional DSL models, even if in principle beneficial to counter accidental complexity, contradict the illustrated pipeline. DSL models, although simpler and more intuitive to create, cannot be readily combined with existing GPL models, and hence also not processed by existing MDE techniques to generate executable code.

1.1 Research Questions and Contributions

This research challenge (namely, the exploration of means to benefit from the fused power of DSLs, AOM and MDE) serves as a motivational foundation for my work.

> *In this thesis I investigate how a combination of DSLs and CORE could become a reality. I propose a plan of action for their integration and collect evidence to determine the viability and effects of corresponding tooling.*

By this rationale, this thesis is structured in three parts, each of which deals with a key research question. This structure likewise aligns with the three main contributions of my thesis.

PART I

Research Question: *Can we define a generic methodology for the integration of DSLs with state-of-the-art AOM and MDE techniques, that effectively eliminates accidental complexity?*

Contribution: With *FIDDLR*, I provide a framework embodying a generic plan of action for the integration of DSLs with CORE. The framework contributes a clear methodology, to tame the otherwise overly complex task of integrating DSLs into reusable concerns.

PART II

Research Question: *Is there evidence for the viability of the proposed framework, for concern creation and reduced accidental complexity?*

Contribution: I provide two novel proof of concept concern implementations. The creation of the sample concerns supports the delineated framework plan of action, and illustrations of their reuse detail how previously complex engineering tasks turn into an intuitive and streamlined modelling activity with minimal overhead.

PART III

Research Question: *Is there conclusive evidence for the reference concerns bringing substantial advantages to Software Engineers?*

Contribution: I provide an empiric assessment of a sample concern toolchain implementation. That is a description of a controlled experiment to measure the effects on software engineers. I detail the experiment layout, methodology, statistical analysis and findings, which conclusively suggest a beneficial impact on the given software engineering task.

1.2 Detailed Thesis Outline

In more detail, the objectives and contribution of the individual parts are as follows:

Part I

In the first part, I investigate how DSLs are best integrated with CORE, in a reusable manner. As initially stated, the challenge lies in any MDE language transformations being tied to the languages they operate on. Therefore, introducing a new DSL disrupts the process. The goal of the first thesis part is to identify where in the existing CORE pipeline DSLs are best integrated, so effects on the existing pipeline are minimally intrusive, as well as to formally define all pipeline adjustments required for patching. In this part I first recapitulate the required building blocks (Chapter 2), followed by an in-depth discussion on the challenges associated with model weaving of DSLs (Chapter 3). The main contribution is the formal definition of *FIDDLR*, a *F*ramework for the *I*ntegration of *D*omain-Specific Mo*D*elling *L*anguages with concern-oriented *R*euse (Chapter 4). *FIDDLR* provides clear design instructions on how to *integrate and reuse existing MDE, AOM/-CORE and DSL tooling* when creating reusable software artifacts, or concerns. Once defined, a *FIDDLR* concern leverages reuse, by allowing the concern user to apply the CSL (that is, a tailored, concern-internal language) concepts throughout the model reuse process, which streamlines reuse and mitigates accidental complexity.

Part II

The second thesis part serves as proof of concept (PoC) validation of *FIDDLR*'s viability. On this behalf, I identify and investigate two representative engineering activities, that is, activities which reflect industrial relevance but are challenging because of inherent simultaneous operations at multiple levels of abstraction (Chapter 6/7). More details on the selected activities follow in the next paragraph. The PoC methodology is nearly identical for both sample activities:

- I first present the traditional engineering approach and highlight why it showcases implicit abstractions, hindering a straightforward, manual task realization.
- Afterwards, I delineate how *FIDDLR*'s guidelines enable the guided creation of a tailored, CSL-enabled concern. That is to say, I demonstrate how the framework specifies the main steps toward an MDE solution package, specific to the sample activity in question. This solution package, or concern, will contain a novel, internal language, which is intended to reify expert knowledge at the moment of concern reuse.
- The PoC then spans out to detailed illustrations of how task solving occurs when reusing the produced sample concern. Note that an essential part of this step is the reuse of the concern provided CSL, which reifies expert domain knowledge. This step illustrates how the activity's initially mentioned abstractions are brought to an intuitive level, allowing a concern user to explicitly bridge from technical to conceptual solution aspects.
- The degree of intuitiveness of a given solution is not easily quantified. We therefore apply an action metric, to measure by how far the targeted engineering activity is simplified by concern reuse.[2] The outcome of this assessment serves as the first evidence of a beneficial effect of the proposal.
- To this point of the report, the presented PoC occurred exclusively on paper, that is we conceptually defined *FIDDLR* and how it manifests into derived concerns. I implemented one sample concern down

[2]Part three of this thesis is entirely dedicated to empirically assessing CSL effects.

to the code level to amend the theoretic considerations by a practical validation. That is to say, I modified the existing CORE reference implementation TouchCORE, so it supports *FIDDLR* adherent concerns. I then followed the *FIDDLR* 's guidelines to derive a fully operational reference concern implementation. In vast parts, the suggested methodology proved one-to-one applicable. In a final "*Lessons Learned*" section I recapitulate the insights gained throughout the actual concern implementation, notably regarding the complexity of the individual *FIDDLR* phases.

With exception to the last point, all the above PoC steps were pursued for both of the following two sample software engineering activities.

- *Conversion of legacy APIs to REST*: According to Postman's 2023 *State of the API Report* [Pos23b], a survey with 40,000 developers and API professionals, 86% of all cloud APIs set on the *REpresentational State Transfer* (REST) paradigm [Fie00]. The widespread acceptance stems mainly from the rise of Micro-Service Architectures (MSA), where REST is used as de-facto standard for inter-service communication. In light of this industrial relevance, the shift from legacy systems to RESTful services is a lasting trend [DGL+17]. In practice, the migration is hindered by the underlying complex technological stack, but also misunderstandings of the paradigm itself [FTE+17]. I deem the conversion of legacy code to REST a perfectly representative engineering activity. It showcases inherent technological and paradigm complexity, rendering it a legitimate candidate for a CSL-enabled solution package.

- *Securing access delegation with OAuth2*: Likewise in the context of modern service architectures is the concept of resource ownership and access delegation. Modern services are built with collaboration in mind. That is to say, service APIs are designed to support secure interaction with third-party services. Almost every online service, from Spotify to the Amazon Marketplace, comes with a notion of resource ownership. If desired, a resource owner can securely grant third-party services access to their resources, i.e. they can enable access delegation without sharing their credentials. The standard protocol for this inter-service scenario is OAuth2, supported by most established online services. The migration of an unsecured service to supporting access delegation is a

conceptually challenging enterprise, for the security concepts stand orthogonal to existing technical service details. This makes API securing with OAuth2 an interesting second study object, for expert security knowledge can greatly guide the conversion process.

As part of the two PoC concern implementations, I also crafted two novel sample CSLs, that is to say, two DSLs intended for use within a given concern. These languages are by themselves interesting study objects, notably on behalf of the discussion of whether there is a conceptual difference toward ordinary DSLs. In Chapter 8, I present two strong arguments that speak in favour of a clear categorical separation and illustrate them on the example of our two derived sample CSLs.

Part III

In the third part, I collect and assess empirical evidence to determine the general viability of CSLs or the combination of DSLs and CORE for the measurable effects for software engineering practitioners. Although the two previous parts provide honest insights into the theoretic functioning and reuse of CSLs and derivative concerns, lab internal research always bears a risk of unconscious bias. An efficient means to fairly evaluate viability is to conduct a controlled experiment with human test subjects. Precisely, I sought to better understand the advantages and disadvantages of using a CSL for addressing a specific concern during software development. I chose REST as a context for my experiment, in part because of its high industrial significance, but also because my previous work had already produced an operational concern reference implementation.
I detail the conducted study, where I asked 28 developers to convert existing applications to expose their services over a REST interface (also a process referred to as "*RESTification*"), once manually and once using a CSL specifically designed for that purpose. I then measured the time required for task completion and tested the produced code for correctness. My quantitative analysis shows that, regardless of declared skill level, developers using the CSL were on average faster and produced solutions of higher quality. Nevertheless, the detailed, qualitative developer feedback confirms sustained skepticism of using CSLs, stemming from a perceived loss of control.

Part IV

The thesis concludes with a final recapitulation, inciting a contextual discussion of the individual parts' findings. I summarize the initial ratio and associated challenge and then delineate how *FIDDLR*, despite perpetuating technical challenges, is a solid foundation. The framework efficiently guided the design and implementation of sample CSL concerns. As such, the framework's viability has been demonstrated both on paper and in practice. Action metric assessments of the sample concerns suggest a superiority of the MDE approach, compared to manual code evolution techniques. This trend is conclusively confirmed by the empiric user study, which furthermore quantifies the effects on conversion speed and correctness for the sample RESTify activity. A side effect of the study is a better understanding of factors that make developers at once recognize the merits of MDE, but keep them reluctant to adopt corresponding tools into their everyday developer activities. The latter are valuable insights for future improvements of toolchain implementations. In summary, this thesis provides a solid foundation for the development of a new generation of MDE tooling, employing a clear plan of action and sound evidence for positive effects on common software engineering goals.

Part I

Concern Specific Languages and their Integration in the Model Driven Engineering Pipeline

In this first thesis part, I present the building blocks for combining DSLs and CORE. This leads to the definition of FIDDLR, a novel MDE framework to guide the implementation of corresponding tools with clear, separate tasks.

2

Background

In this chapter, I present the design principles and technologies that serve as building blocks for the first thesis part. For now, the focus lies on their contribution to Software Engineering, although throughout the remainder of the first part of the thesis I delve into their combining. In the following sections, I am first interested in *Separation of Concerns* (SoC) and *Planned Reuse* as engineering objectives. Then I present an overview of the MDE process pipeline, Domain Specific Languages, and Aspect-Oriented Modelling on the example of Concern-Oriented Reuse.

2.1 Separation of Concerns and Reuse

SoC has been identified early on as one of the main mechanisms for tackling complexity during software development [Dij76]. The term was coined by Dijkstra and refers to the ability to temporarily focus one's attention solely on one development concern or issue. There are various means to achieve SoC, but important ones are e.g., encapsulation and information hiding, which are likewise key principles of the Object-Orientation paradigm.

Reuse is simply the process of creating software systems from existing software artifacts rather than creating them from scratch [Kru92]. There is a clear distinction between opportunistic and planned reuse. Opportunistic reuse refers to extracting models or code artifacts from existing projects,

where notably these projects had not been created with reuse in mind. That is, the act of reusing proven project components takes place in hindsight. Planned reuse on the other hand refers to the act of intentionally creating models or programming artifacts for reuse. This is for instance the case with software libraries, that were not created to be viable in isolation, but from the start intended for invocation. In this thesis, we are mostly interested in *planned reuse*.

2.2 MDE, Modelling Languages and Processes

Model-Driven Engineering (MDE) [Ken02, Sch06] is a unified conceptual framework in which the whole software life cycle is seen as a process of *model production*, *refinement* and *integration*. Commonly this also implies that code is not written by hand, but the final product of such a process. The individual models are built to represent different views of a software system using different formalisms, i.e. modelling languages. That is to say, there is a palette of languages to choose from when creating a model. Notably, this means that for each model the language is chosen in such a way that the model concisely expresses the properties of the system that are important at the current level of abstraction. In paraphrasing terms, this means that a good language choice allows one to accurately express what is of interest, with fitting expressiveness and minimal overhead.
Concerning the aforementioned refinement, this means that during development, high-level specification models are extended or combined with other models to include more solution details. That is, as models become less abstract, they pertain to more implementation specifics. The manipulation of models along this refinement is achieved by utilizing model transformations. In summary, model refinement and integration continue until a model or code is produced that can be executed.

When it comes to a palette of commonly used modelling languages, a typical MDE process makes use of one or several *General Purpose (Modelling) Languages* (GPLs). The left side of Fig. 2.1 a) depicts typical software development phases found in object-oriented, model-driven development methods for various development phases. Note that the terms *Requirement Phase*, *Design Phase* and *Implementation Phase* represent the aforementioned levels of refinement, where abstractness is gradually substituted by implementation-specific details.

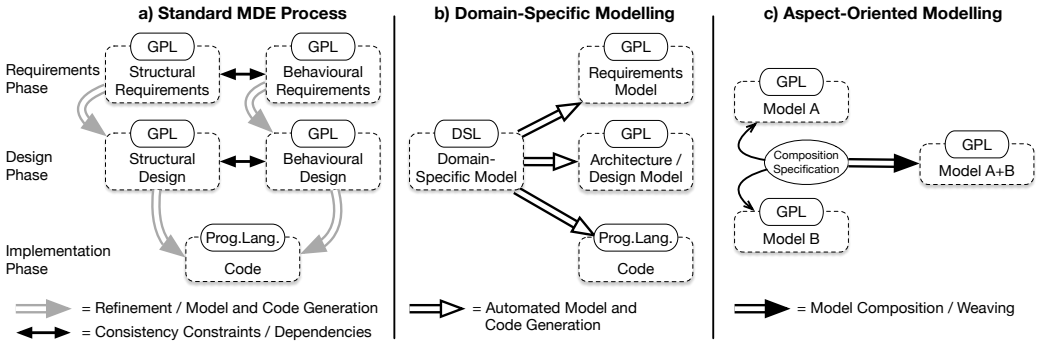

Figure 2.1: MDE, DSL and AOM at Various Levels of Refinement

Note that the crafted models do not co-exist in isolation. At any given level of abstraction, consistency constraints ensure that the different models form coherent views of the system. This is an important detail since otherwise, coexisting models would likely contradict, hindering any subsequent refinement. In Fig. 2.1 a) these constraints are depicted with black double-ended arrows. Ultimately, code generation is used to generate a significant part of the object-oriented implementation from the design models. That is, the outcome of the process is compiling or interpretable models/code. Guidelines for refinement, and model transformations that implement partial refinement and code generation are depicted with thick grey arrows in Fig. 2.1 a).

To put things into relation with the initially stated Separation of Concerns principle: SoC is at the heart of MDE. Every model created is an abstraction of the system under development – unnecessary details (considering the current level of abstraction) are omitted. When establishing a model, the most appropriate modelling language is used, focusing the attention of the modeller on the current properties of interest. Finally, each model describes the system under development from a different point of view, and can therefore focus on a different development concern.

2.3 Domain-Specific Modelling

The previously described MDE process sets entirely on GPL models for all stages of modelling. Once more, I'll use paraphrasing to illustrate the nature of these models. GPLs are languages that comprehend the standard concepts required for a proper description of object-oriented systems at var-

ious levels of abstraction. GPLs can be seen as a one-size-fits-most language palette, and as long as the system under description adheres to standard requirements, design and implementations, GPL instances like the Unified Modelling Language (UML) [OMG17] are a meaningful choice.

Domain-Specific Languages (DSLs), on the other hand, stand in stark contrast to this general-purpose philosophy. DSLs have been used in computing since the early days [vDKV00], are typically small, and focus on a particular aspect of a software system or the development process. A popular DSL example is "*Cascading Style Sheets*", as a textual language to describe visual properties of elements defined in a markup language document. It is not uncommon for DSLs to be used in combination with general purpose languages (GPLs) to develop larger software systems [Fow10] and amend the fractions that are easily covered by GPLs by additional, more specific models. Some argue that well-designed DSLs are much easier to work with than a general purpose language [Gra07], as with a DSL, a developer maximally focuses on the particular task at hand, and the problem can be dealt with using concepts of the problem domain that are reified in the language. For example, a DSL for mobile phones would allow modellers to specify high-level abstractions for the user interface, as well as lower-level abstractions for storing data such as phone numbers or settings [MV10].

A strong argument for the use of DSLs is their capacity to minimize accidental complexity [AK08]. Accidental complexity arises out of a mismatch of modelling language and modelled matter. While GPLs such as UML mostly cover the typical structural and behavioural modelling needs for *software development*, their general purpose nature can imply a (sometimes significant) semantic gap between a specific application domain and the concepts offered by GPLs. This gap can be bridged with DSLs [Gra07], which in turn means their use bears the potential to significantly improve productivity.

An important mention that will play a significant role throughout this thesis, is that most DSLs come with the tool support that allows combining the DSL-based artifacts with the remainder of the code base and software system. This can for instance be the case via DSL specific transformations that allow translating DSL models into GPL or code counterparts at various levels of abstraction, which is illustrated in Figure 2.1 b).
As previously mentioned, in MDE, model transformations and consistency constraints are a key concept, therefore the introduction of novel languages implicitly also comes with a requirement for additional tools. This is notably

relevant in the context of *Reuse*, i.e., the main unit of reuse being a modelling language: A modeller using a modelling language is reusing knowledge of the language engineer when building models by instantiating language concepts. To the aforementioned transformations and language constraints, this means language reuse is only readily possible if the required tools are likewise built for reuse.

Finally, despite many reports on the potential of DSLs [KMB+96, Wil03, KGCM18], the use of DSLs is not widespread in practice, as many developers favour a known workflow in familiar development languages and tools over the DSL-based alternative [GFC+08].

2.4 Aspect and Concern-Oriented Reuse

In the previous sections on MDE and DSLs we have mainly considered wholesome models, that is, models that in isolation define a cohesive and viable unit. In contrast to this practice stands Aspect-Oriented Modelling (AOM). In AOM, a modelling language is augmented with advanced language features that enable the modularization and composition of model fragments. Model fragments are models that are not necessarily viable in isolation. That being said, there is an imminent necessity for a means to bring back together what is not viable in isolation. This issue is addressed by a so-called model weaver. A model weaver is a special model transformation that takes as an input two models and a composition specification and produces a new *composed* output model in which the two input models have been merged, see Fig. 2.1 c). Note that weavers only operate on specified input models of the same language. This detail will play an essential role in the remainder of this thesis part.

AOM does not guide how models are best decomposed or recombined. As a result, AOM is at once a powerful, but also overwhelming concept. Notably concerning maximized planned reuse, AOM does not provide clear instructions on how to best operate on given models. This problem is addressed by Concern-Oriented Reuse (CORE) [AKM13]. CORE is an approach based on AOM that streamlines model reuse by encapsulating model fragments inside a reusable unit called a *Concern*. The idea is that a vast collection of concerns serves as an off-the-shelf solution library for most common modelling challenges, effectively eliminating the need to reinvent the wheel for common modelling patterns.

In CORE, the reuse process, where a concern user accesses the embodied concern knowledge, is a sequential process of three phases. The interfaces for these stages must be anticipated and created by the engineer designing the concern (also called *Concern Designer*). In detail, a concern designer must provide three interfaces to guide concern reuse [KMA+16]:

1. The *Variation Interface* (VI) exposes the different variants of the reusable entity with a feature model, and the impact of each variant on high-level system qualities with an impact model. The latter can e.g. provide estimations on global design criteria like throughput, energy consumption, or system security.

2. With the *Customization Interface* (CI) the concern designer exposes the generic entities in the concern that have to be adapted to a specific reuse context.

3. Finally, the *Usage Interface* (UI) defines how the functionality encapsulated by a concern may be used. In this step, the concern user creates the link between the customized model and the application context.

We can readily illustrate this process with the example of a concern allowing for the reuse of a design pattern. Consider a common situation during detailed design where some data is modified, and as a result, several graphical objects that visualize the data have to be updated to reflect the state change. This feat is easily accomplished using the *Observer* design pattern [GHJV95].

1. The VI for a concern encapsulating the *Observer* design pattern would take the form of a feature model that exposes different variations of the pattern to the concern user, e.g., the *Push* or *Pull* variant. The structure of the *Observer* design pattern can be well expressed with a UML class diagram fragment. The behaviour, namely that all registered *Observer* instances should be notified when the state of an observed *Subject* changes, can be described with a UML sequence diagram fragment.

2. In this example, the CI would require the concern user to map the *Subject* and *Observer* classes to the appropriate application design classes and to do the same for their respective *modify* and *update* operations. Throughout this thesis, I often illustrate the aforementioned steps on

the example of TouchCORE, a state-of-the-art reference implementation of CORE [Kie23]. TouchCORE renders the described process a convenient procedure. Eligible subjects can even be loaded dynamically from existing software artifacts.

3. The UI finally is constituted of operations that allow an *Observer* instance to subscribe to or unsubscribe from a *Subject*.

In summary, by giving a *Concern User* access to these three interfaces, CORE streamlines the reuse process by allowing a concern user to:

(a) Choose a desired variant from the VI.

(b) Adapt the chosen models to the specific reuse context with the CI.

(c) Use the structure and behaviour encapsulated by the concern exposed in the UI.

Note that behind the scenes, the described process heavily sets on AOM, that is specification of partial models, composition specifications and subsequent model weaving. The information provided by the concern user (i.e., the selected features designate a set of realization models, in our Observer example UML class and sequence diagram fragments., the customization mappings and the usage dependencies) constitutes model fragments and a composition specification. A weaver then combines the model/code fragments, using the provided composition specification. The outcome of the reused concern reflects the application context integrated with the selected concern features and customization. In the last step, using standard MDE transformations, the woven models can be used as input for code generation.

Although the described CORE reuse process combines AOM (for partial models and model weaving) and MDE (for model transformations such as code generation), CORE is not free of limitations. Notably for crosscutting concerns, that is, concerns that do not align well on one level of abstraction, the resulting concern models showcase severe accidental complexity. As previously mentioned, while DLS are the standard response to this challenge, integration of additional languages into the existing MDE/AOM pipeline is not trivial. CORE lacks a generic and versatile plan of action to mitigate accidental complexity.

2.5 On Building Block Combinations

All of the concepts presented throughout this chapter bear certain advantages to software engineering. It seems only natural to strive for an uncompromising combination of all their contributions, lifting software engineering to the next level. However, this is not easily done, for the characteristics of one concept often contradict the functioning of another. In the remainder of this first thesis chapter, I present the challenges associated with combining building blocks and present a plan of action to overcome hindering factors.

3

Polyglot Weaving

In the previous chapter, I presented various MDE-related techniques to support Separation of Concerns and Reuse, which are generally considered beneficial factors to SE. A natural follow-up question is to which extent these approaches can be combined as building blocks, to likewise combine their advantages. I will now briefly delineate the potential of such a combination, and then reason why the challenge of combining multiple modelling languages represents a key hurdle to this enterprise. Throughout the chapter, I will also refer to this challenge as *polyglot weaving*. The chapter closes with a solution sketch on how to overcome the issue. This sketch will serve as a foundation for the formal and generic framework proposal discussed in the next chapter.

3.1 Combining Building Blocks

Common motivation to all building blocks presented in the previous chapter is their contribution to *Separation of Concerns* and *Planned Reuse*, as fundamental SE drivers. Unfortunately, there is no clear favourite of the presented techniques, for every building block, be it MDE, AOM/CORE or DSLs comes with its advantages and downsides. As such, one could wonder if we could simply combine those building blocks, and ideally obtain a synergistic solution where advantages are combined, and individual drawbacks compensated. More precisely, one would hope for a solution combining the

SoC power of model-driven abstractions and life-cycle refinements to leverage language-based SoC, with aspect-oriented modelling techniques and concern-oriented reuse to foster SoC for crosscutting concerns as well as streamline reuse. Ideally, on top such a solution would also mitigate any accidental complexity with DSLs. The outcome of such a fictitious combination would be a software development tool that places focus on one concern at a time, maximizes fast reuse of proven solutions, and reaches all that at an intuitive and to-the-point level of language expressiveness.

Unfortunately, to date, there is no such approach or tool. Because of how they function, the individual building blocks are not so easily combined. But that does not necessarily imply a general impossibility.
In the following, I reason why CORE is a meaningful starting point for exploring the aspired combination, and illustrate the key conceptual challenge that needs to be solved. Afterwards, I briefly sketch how CORE's existing AOM and MDE pipeline could be modified and extended, to conceptually solve building block compatibility.

3.2 The Polyglot Weaving Challenge

I argue that CORE is a meaningful starting point for the aforementioned endeavour. CORE as an AOM derivative is built with reuse in mind. Furthermore CORE emphasizes SoC principles: the relevant properties of a CORE concern are expressed at the appropriate level of abstraction using the most appropriate GPL. Additionally, CORE showcases great compatibility with MDE, and selectively already makes use of MDE transformations, namely code generation. Figure 3.1 illustrates how classic CORE conceptually combines key advantages of AOM and MDE in a GPL-oriented modelling pipeline.

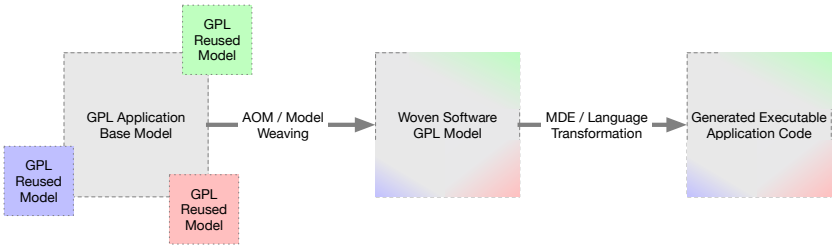

Figure 3.1: Simplified Illustration of the Classic CORE Pipeline

However, for concerns that do not align well with GPL concepts, the standard way of customization and usage offered by CORE introduces significant accidental complexity. For example, imagine a situation where the design of an application is modelled using class-, state- and sequence diagrams. Imagine now a *Workflow* concern that can be used to define and execute workflows that are constituted of interdependent and potentially concurrent activities. Neither state nor sequence diagrams are well suited to model workflows. While those models can be used to design a workflow execution engine, the customization of this workflow engine design would be very difficult for a concern user, who would have to understand the internal design details of the engine.
In the context of the classic CORE pipeline presented in Figure 3.2 this would correspond to the outer GPL models growing overly large and complex (for what they express) due to a mismatch of language concepts. The Xes indicate unreachable or hindered pipeline steps. That is, the entire reuse process is hindered by the accidental complexity of input models. For a concern user that means running through the VCU steps becomes overly complex, the contrived models distract from the relevant decision-making points, and in the worst case counteract the aspired CORE advantages.

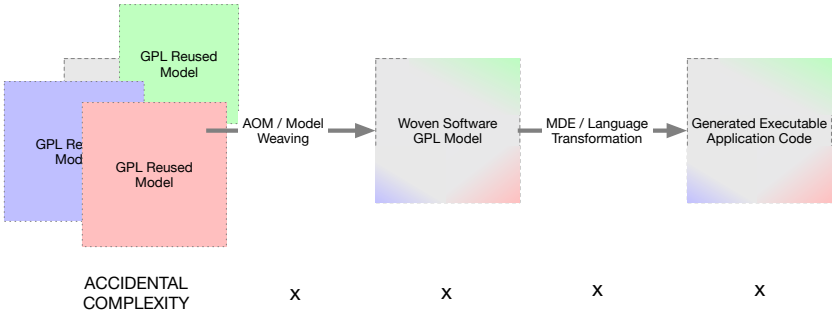

Figure 3.2: Illustration of Accidental Complexity in Classic CORE

Further examples are easily found, for in essence the SoC power of MDE is limited whenever it comes to development concerns that do not align with the levels of abstraction of commonly used GPLs. This is notably the case for development concerns, e.g., *Security*, which need to be considered not only during the requirements phase but also during architecture, design and implementation. As we will see later in this thesis, addressing security properly requires dealing with security-related structure and behaviour at all phases of

development. Hence, if only GPLs are used, security-related model elements end up scattered across multiple models. In this case, the use of modelling languages that are not aligned with the development concern in question introduces what is called *accidental complexity*.

As discussed in the background section, DSLs have the potential to address this drawback, for custom languages can define tailored features that allow a modeller to express properties relating to any level of abstraction of software development for the targeted domain. It would therefore perfectly make sense to allow CORE concerns to reify solutions and expert knowledge utilizing tailored languages, effectively overcoming the aforementioned conceptual mismatch. In principle, this would allow the definition of concerns that are more convenient to use, and that reflect more accurately the nature of the unit of reuse and the path toward its contextual integration.

Unfortunately, this combination exposes a conceptual incompatibility. Although DSLs are a powerful means to mitigate accidental complexity, they are not readily compatible with CORE's reuse pipeline. CORE, as an AOM derivative, sets on model weaving, that is combining several models based on composition specifications. The weaver consumes the concepts of either input model, along with a composition specification, to create a new model. However, this only works as long as both input models are formulated in the *same* input language - in a modelling context this translates to: "weaving is only possible if the inputs adhere to the same metamodel".
The moment we switch one GPL specification for a DSL, weaving is no longer possible, regardless of how efficiently accidental complexity has been mitigated. Figure 3.3 illustrates this incompatibility in the CORE pipeline.

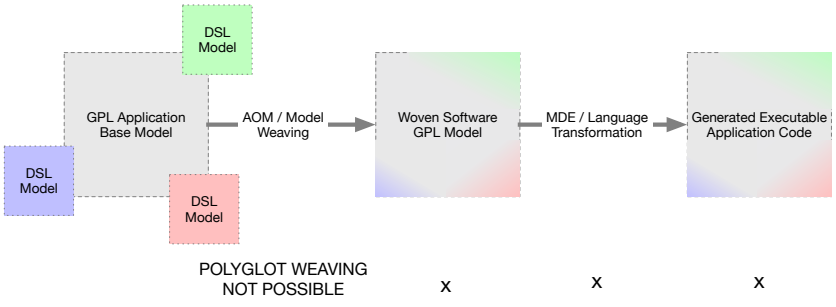

Figure 3.3: Illustration of Polyglot Weaving Challenge

3.3 Solution Draft

The described incompatibility of model weaver and auxiliary DSLs is hardly a discovery. Unless DSLs are used for purely illustrative purposes, they necessarily require tailored model transformations to reestablish compatibility with existing modelling languages, notably GPLs. In the spirit of CORE, the nearby solution is, therefore, to not only ship concerns along with tailored languages but also include the transformers needed to restore general compatibility. This way CORE concerns maintain their role as first class citizens when it comes to planned reuse. They provide a cohesive package of models, language and transformations. In the following, I refer to DSLs, specifically built for reuse in CORE concerns as *Concern-Specific Modelling Languages,* or CSLs. Throughout this thesis, we will see multiple CSL representatives and the thesis also contains a discussion on conceptual differences between DSLs and CSLs.
A provision of CSLs, alongside tailored model transformations to ensure compatibility with GPLs, allows the following modification to the classic CORE pipeline: Instead of directly weaving concern-derived models with contextual GPL models, the pipeline begins with concern-provided CSL to GPL transformations that ensure all weaver inputs are compatible. This step is illustrated in Figure 3.4

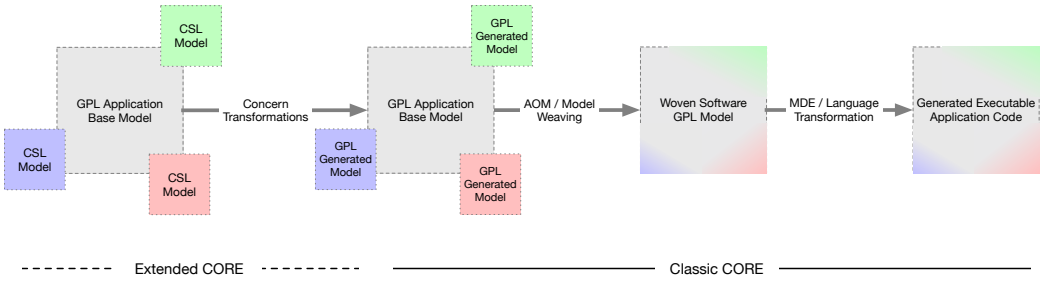

Figure 3.4: Extended CORE Pipeline with Concern Transformations

A charming observation of the solution sketch is that from the moment of model weaving, the remaining CORE pipeline remains untouched. Notably, this means that the proposed pipeline extension is fully compatible with existing CORE reference implementations. This effectively means that the proposal makes perfect use of the initially discussed AOM and MDE steps, effectively making good use of all presented building blocks.

Note that intermediate GPL models may still suffer from accidental complexity. Since we translate lean DSL models to standard GPL models, there is a substantial risk for the intermediate GPL models to be overly complex and hard to consume by humans. However, since this is only an intermediate step in the fully automated CORE/MDE pipeline, this should not be considered an issue. Even if overly complex, a concern user would never be exposed to such intermediate models.

On the presented, abstract level, the solution sketch seems justifiable. However, considering integration with CORE in more detail reveals an abundance of variation points, and it is far but clear how DSLs and their transformations are best integrated with concerns for optimal reuse. A related challenge is the various levels of abstraction contained in the CORE pipeline, which likewise means the transformed DSL models may integrate with GPL models of various stages. In the remainder of this first thesis part I therefore delve deeper into the details of the presented solution sketch, to explore how DSL integration with CORE is best formalized to a generic and reusable plan of action, taking shape in the form of a framework that integrates MDE, AOM and DSLs.

4

FIDDLR

Throughout this chapter, I will gradually detail the previously sketched integration of MDE, CORE and DSLs. I describe how the introduction of CSLs affects perceived concern reuse, and respective implications for Concern Design tasks, notably taking into account models at various stages of MDE refinement. Ultimately, the contribution of this chapter is to extract a step-by-step plan of action, allowing the creation of concerns that come with their own CSL. That is, I will present a framework with clear separate tasks assisting concern design with maximal reuse of existing CORE and MDE concepts.

4.1 CORE-MDE Pipeline Extension Details

A central element of this chapter is the aforementioned pipeline extension. To discuss and understand the implications of such an extension, we need to delve into the integration with the existing CORE pipeline. CORE is already by nature complex, and extending the pipeline adds additional complexity, which is why the corresponding Figure 4.1 is at first overwhelming. Therefore, I first present preliminary guidance on how to read the illustrative figure. Throughout the remainder of this chapter I will then gradually cover individual elements in more detail.

4.2 Concern Reuse

Figure 4.1 has two dimensions. The horizontal axis, from left to right, represents pipeline progress. On this behalf, the figure is strongly oriented towards the previously provided sketch in (Figure 3.4). The left column "Application" illustrates the standard MDE stages of model refinement for a sample application. A concern then makes use of these contextual models to amend them with concern-provided expertise in the form of CSL models. The integration is achieved with the help of AOM and MDE techniques, until at the end of the day (on the right side) we have obtained the desired models, combining context and concern.
What is new, compared to the previous solution sketch, is the vertical dimension. The three layers from top to bottom represent different levels of MDE refinement, namely from Requirement to Architecture & Design, to Code.
Throughout the remainder of this chapter, I will now gradually describe all pipeline elements. I begin with the concern user perspective, that is, how concern reuse is experienced. Afterwards, I proceed to the concern designer's perspective. By the end of the chapter, this allows us to have a good understanding of not only how extending a CORE pipeline by CSLs affects the daily business of concern user and designer, but also the extraction of hands-on concern design guidelines, in the form of a dedicated framework.

From the perspective of a concern user, the main difference lies at the beginning of the CORE pipeline. Instead of applying the VCU dimensions exclusively through GPL models, the concern user gains access to novel language elements, which are tailored specifically to the given concern's reuse. The benefit of these new elements is that it allows a concern user to accurately access and reuse expert knowledge at a meaningful level of abstraction, which effectively mitigates accidental complexity. By using such language elements, the standard CORE reuse process [KMA+16] is streamlined for the concern user. In Figure 4.1, the use of the custom language is illustrated by the blue box, representing a user-defined model in the concern's CSL.

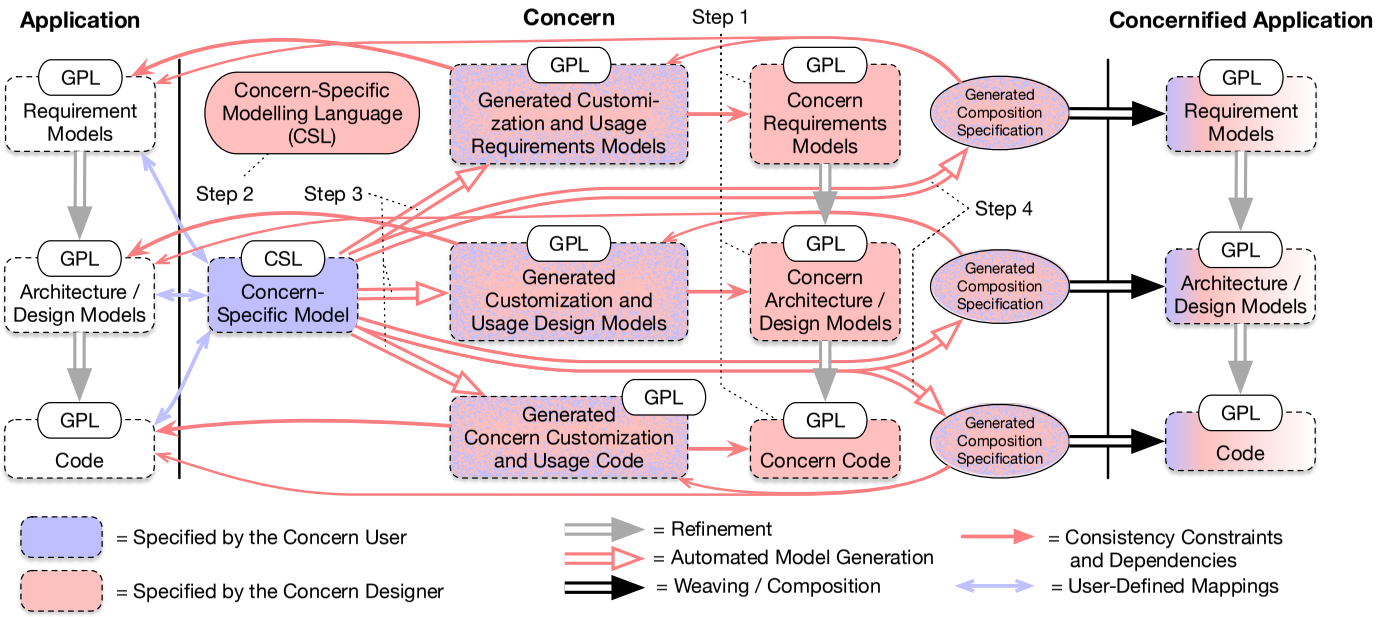

Figure 4.1: Details of the CORE Pipeline Extension to Support CSLs

Once a CSL model is created, it must be placed in context with the base application. This is achieved via model mappings that may occur at all levels of MDE refinement. Note that the mapping phase corresponds to the customization and usage dimension in classic CORE, as it involves the linking of the appropriate model elements from the created CSL model to model elements of the application. In Figure 4.1, these mappings are indicated as double-ended blue arrows. Not all levels are necessarily represented in every concern reuse, or in other words: which mappings are needed highly depends on the concern's nature. Throughout the thesis, we will see concern samples that only cover a subset of all possible user-defined mappings for the three indicated MDE levels of refinement.
In short, the main difference to classic CORE is that using the extended pipeline, concerns come packaged with an embedded CSL, and concern users access the newly available language concepts to more accurately express concern-related properties and how they are integrated into the context. As for concern reuse, no more action is required once a custom concern model has been created and mapped, for the pipeline remainder is executed in an automated way.

4.3 Concern Design

In CORE, the unit of reuse is the *concern*. In the spirit of planned reuse, integration of a concern should therefore be as straightforward an activity as possible. The previously presented perspective of a CSL concern user, who applies a custom language to accurately access and apply tailed concepts, is an illustrative example for streamlined planned reuse.
However, while reuse should be as convenient as possible, the crafting of an

easy-to-reuse concern is by nature a complex task. If custom languages are to be part of the process, then concern creation becomes even more complex. At the end of the previous section, I stated that from a user perspective, the execution of the required model transformations, weaving and code generation are performed in an automated way. In this section I describe what tooling is required, and therefore must be anticipated by a concern designer, to support such reuse convenience. Regarding the initial Figure 4.1, we are now zeroing in on the continuation of the pipeline, right after the elements described for Concern Reuse. As a whole, the automatic processing of user-provided CSL models and mappings requires four concern-provided components, each of which I now describe in more detail. To visually distinguish the Concern Designer's activities from the ones of the Concern User, the components that have to be provided by the concern designer in Figure 4.1 are visualized in red. Since the provision of these components falls into the responsibility of the Concern Designer, I also refer to them as *activities*.

4.3.1 Realization Models

The first activity is identical to the classic CORE pipeline. As presented in 2, concerns come in variants, and for each variant supported, a concern designer must define the associated off-the-shelf models, describing the concern's static components. These are called *Realization Models* and their defining corresponds to *Step 1* in Figure 4.1.
The Figure depicts Realization Models at all three MDE refinement stages: Requirement, Architecture & Design, and Code level. However, which exact models are needed depends on the MDE process being used, and on the nature of the concern. Some concerns crosscut at all levels of abstraction, e.g., *Security*, and therefore such concerns contain many realization models. Other, more solution-oriented concerns, notably the ones later presented in this thesis, might primarily affect lower levels of abstraction, e.g. introducing boilerplate code required to initialize a specific implementation platform. The concern designer who has to perform this step should be a developer with expertise in the implementation of the concern, in collaboration with an expert of the GPL modelling languages used in the MDE process. Once more, note that this activity is unchanged compared to the classic CORE pipeline, for all Realization Models are GPL models.

4.3.2 CSL Design

The main interest of the aspired pipeline extension is the ability to streamline the reuse process with an additional, tailored language, that simplifies customization and usage of a concern. However, for such a language to be available to a concern user, the language must first be defined. Since the language is inherently tied to the concern it seeks to streamline, and since it is shipped as part of the concern, its definition falls under the responsibility of the concern designer. In Figure 4.1, this is represented by *Step 2*, the Concern-Specific (Modelling) Language (CSL) definition. This step should be performed by a DSL expert collaborating with the concern domain expert, who would specify the language's abstract syntax in the form of a metamodel. Part of this step is also a definition of which CSL model concepts can be related to existing GPL model elements. These mappings are also referred to as "*Model Element Mappings*" (LEMs). Furthermore, model editing operations should be defined for consistently manipulating model instances.

4.3.3 CSL Model to GPL Model Transformation

The concern user's CSL to GPL mappings exemplify the aforementioned polyglot weaving challenge. In pursuit of rendering concern reuse as convenient as possible, we allowed the concern user to directly map concepts of different model types, that is, we allowed the concern user to map from their concern CSL model to the contextual GPL models (blue-double-ended arrows in Figure 4.1). As previously discussed, the CORE weaver cannot readily consume these mappings, for two reasons:

1. The weaver cannot interpret the CSL model, for it is not familiar with the language definition.

2. The weaver cannot combine models, for they do not share a common metamodel language definition.

Therefore it is up to the concern designer to include the required tools within the concern, that is, as part of their concern design activity. The first step is the provision of a model transformer to convert CSL models into GPL models. As a reminder, the GPL models will most likely showcase accidental complexity, which is why from a user perspective direct GPL modelling is to be avoided. Step 3 in Figure 4.1, red arrows originating from the CSL model illustrate the model translations, using the transformer provided by a concern

designer. Note that possibly multiple transformers are needed, to cover all levels of MDE refinement, which is why the figure showcases arrows to all three MDE levels. The produced GPL models are illustrated in speckled red and blue, for they contain combined expertise of concern user (CSL model) and concern designer (realization models and model transformations). This step should involve a model transformation expert, possibly again in collaboration with a concern implementation expert.

4.3.4 CSL Mapping to Composition Specification

The previously described CSL to GPL model transformers addressed the underlying polyglot weaving challenge. However, the described transformations are incomplete. The user-provided input was not only a CSL model, it was a *mapped* CSL model. Therefore it is insufficient to only translate the CSL model into GPL models. The transformations must also deal with whatever mappings are connected to the original CSL model. Provision of these mapping translators is indicated as *Step 4* in Figure 4.1. The outcome of this second set of transformations then serves as composition specification for subsequent model weaving. In Figure 4.1, this output is illustrated in speckled blue and red, for once more it contains information provided by the concern user (mappings) and the concern designer (mapping translations).
Note that once more, the translated mappings might need to be provided for multiple levels of MDE abstraction. The concern realization expert and the MDE expert need to decide at which level of abstraction the concern-specific model is best composed with the application's realization models. For example, some of a concern's behaviour might best be composed at the code level, while other behaviour can better be composed at the level of state charts or sequence diagrams. A model transformation expert then designs a transformation that, given a CSL model and mappings provided by the user as input, produces composition specifications for the customized GPL models.

In summary, to support the convenient reuse of concerns with an integrated CSL, the concern designer has to provide four concern components. But notably these steps can be fulfilled by individual experts in their fields. For example in 1) a DSL expert would collaborate with the concern domain expert to define the concern-specific language (meta-model and language actions). Then 2), the concern domain expert and the MDE expert (i.e., the expert in GPLs, such as UML, that are used in the MDE process and for

code generation) would figure out at which level of abstraction the concern-specific model is best composed with the rest of the application. For example, some behaviours might best be composed at the code level, while other behaviours can better be composed at the level of state charts or sequence diagrams. Finally, for steps 3) and 4), a model transformation expert, again in collaboration with the concern domain expert, would write the model transformations from the CSL model to the chosen GPL models/code, and from the CSL-GPL mappings to the GPL-GPL mappings, if any.

Table 4.1 provides an overview of the respective tasks and which respective experts are best involved per component.

Step	Description	Involved Experts
1	Concern Realization Models Definition	GPL, Domain
2	CSL Meta-Model Definition	DSL, Domain
3	CSL Model to GPL Model Transformer	Model Transf.
4	Mapping to Composition Spec. Transformer	Model Transf.

Table 4.1: Concern Designer Activities Required for CSL Support

With the above steps, we define *FIDDLR*: A *F*ramework for the *I*ntegration of *D*omain-Specific Mo*D*elling *L*anguages with concern-oriented *R*euse.
The contribution of FIDDLR is that it splits the task of designing a concern into smaller, independent steps, namely *concern realization*, *CSL design*, *CSL* $\rightarrow$ *GPL transformation*, and *CSL* $\rightarrow$ *composition specification*. Each step reuses existing technologies whenever possible, thus simplifying concern design and reducing the amount of work required significantly. The *FIDDLR* concern design steps can even be distributed over a team of individual experts in their field.

A closing note is that the convenient use of custom modelling languages often spans out to graphical editors. However, this is not a hard requirement, for modelling can likewise occur in textual models, and in some cases is even the preferable work mode. The *FIDDLR* framework therefore does not incorporate a mandatory step for the design of a graphical modelling editor. Throughout this thesis, I will nonetheless present sample graphical editors for two reference CSL concerns, for illustration purposes.

4.4 Concern Composition

Concerning Figure 4.1, with the two previous sections we have now covered most elements. We've seen how CSL model and mappings are provided by the concern user at the pipeline start.
Taking into account these inputs we've then reasoned which generic steps are required by a concern designer to support and process these inputs, which lead us further through the pipeline, to GPL models that contain the application-specific customization mappings and usage of the concern API, as well as composition specifications that connect the generated models with the application models at each relevant level of abstraction.
So far not covered is the last pipeline step, the actual combining of GPL models and composition specifications to a concernified application. This step notably does not fall into the responsibility of the concern user or concern designer. In the spirit of reuse, the combining of application and concern-derived GPL models, following a composition specification is classic CORE/MDE business, that can be used off-the-shelf as is. In more detail, the composition specifications and GPL models are simply provided as input to the CORE model weavers, which then generate the concernified application, i.e., the GPL models in which the concern-specific and application-specific structure and behaviour have been combined.
This is followed by the classical MDE step that, for the interest of readability, has not been included in Figure 4.1, is the subsequent generation of compilable or interpretable code. Ultimately, the interest of all Software Engineering activities is the execution of the target program. In this case, the generated code ideally reflects both, the context and the reified concern knowledge, and is as such in line with the original program requirements.

In summary, in this chapter, we have elaborated *FIDDLR*, a rigorous framework that defines a four-step action plan to support the general design of CSL-enabled concerns. *FIDDLR* puts forward the idea that DSL technology can be exploited effectively for implementing and applying concerns that do not align well with standard GPL concepts. In particular, *FIDDLR* defines an approach for packaging a DSL with a concern, and as a framework provides clear tasks to integrate the concern implementation with MDE tooling, existing GPL models and code. *FIDDLR* therefore is beneficial for both concern designers and concern users.
The presumed advantage of *FIDDLR* is the ability to construct concerns

with built-in CSL, which in turn allows usage of concerns that otherwise suffer from accidental complexity throughout reuse. Using *FIDDLR* extends the solution space of CORE to concerns that do not align well with GPL concepts.

However, the definition of *FIDDLR* is just the beginning. In the second thesis part, I strive for concrete viability evidence by applying *FIDDLR*'s plan of action to construct and assess reuse with two sample concerns, each showcasing a built-in CSL. Afterwards, in the third thesis part, I extend the assessment with an empirical component. I perform a controlled experiment with humans to determine the measurable effects of a sample CSL-based concern on a representative and relevant engineering task.

Part II

Two Sample CSL Toolchains

In this second thesis part, I apply the previously defined FIDDLR framework to create two novel CSL-enabled concerns: the RESTify and the AUTHify concern. These reference implementations serve as proof of concept for the plan of action provided by the framework.

5

Background

This chapter provides the contextual background for the two sample CSL-based concerns that serve as Proof-of-Concept for illustrating *FIDDLR*'s plan of action. However, before I delve into the details of their design, I will now first present the two engineering activities that the sample concerns address. This background section is also relevant to provide a survey of existing building blocks reused during concern composition.

The two targeted software engineering activities of interest are: "*Exposing existing functionally through a REST interface*" and "*Securing services for access delegation support*". This chapter is structured in two sections: For each activity, I first present the underlying paradigms, followed by evidence and illustrations on the activity's industrial relevance. Afterwards, I provide a brief survey of existing contextual DSL approaches and, where applicable, code generators or code samples, used for those two activities. This chapter lays the foundations for the subsequent definition of tailored CSLs, CORE mappings and code generation strategies, which are the main steps laid out by *FIDDLR*. The actual concern design is then discussed in detail in Chapters 6 and 7, respectively.

5.1 Representational State Transfer

Representation State Transfer (REST) is an API design paradigm, where the actual service functionality is hidden behind a representative abstraction. The API is designed to maintain an illusion of working on file-system-like, hierarchically structured resources, the state of which can be altered with CRUD operations [Fie00] (Create, Read, Update, Delete). Performing such an operation invokes service functionality, which to the outside is perceived as querying or transferring resource state.
Those operations are commonly invoked over HTTP as *Put, Get, Post* and *Delete* requests. Clients interact with a service that is adherent to the REST style (i.e., a *RESTful service*) uniquely over those selectively enabled CRUD operations. Having a RESTful service therefore constitutes a strong layer of abstraction, as it strictly conceals service implementation details. This notably distinguishes REST from simple *Remote Procedure Calls*, where an existing API is exposed as is, and only the communication channel changes.

Throughout the next chapters, I will often use a simple Java desktop application as an illustration. It is a tiny e-commerce like *BookStore* [Sch21] application, which can be refactored to a RESTful service, illustrating the common challenges when re-exposing service functionality through REST. Table 5.1 illustrated the aforementioned REST concepts on the example of the BookStore, that is, one column for the HTTP CRUD operation, one column for the file-system-like location in a resource tree, and finally the original BookStore method represented by this abstraction.

Operation	Resource Path (`/bookstore` prefix omitted)	BookStore Method (Parameters omitted)
GET	`/isbns`	Assortment.getEntireAssortment()
GET	`/isbns/{isbn}`	Assortment.getBookDetails()
PUT	`/isbns/{isbn}`	Assortment.addBookToAssortment()
GET	`/isbns/{isbn}/comments`	Comments.getAllCommentsForBook()
POST	`/isbns/{isbn}/comments`	Comments.addComment()
DEL.	`/isbns/{isbn}/comments`	Comments.removeAllCommentsForBook()
POST	`/isbns/{isbn}/comments/{commentid}`	Comments.editComment()
DEL.	`/isbns/{isbn}/comments/{commentid}`	Comments.deleteComment()
GET	`/stocklocations`	GlobalStock.getStoreLocations()
GET	`/stocklocations/{location}`	GlobalStock.getEntireStoreStock()
GET	`/stocklocations/{location}/{isbn}`	GlobalStock.getStock()
POST	`/stocklocations{location}/{isbn}`	GlobalStock.setStock()

Table 5.1: CRUD Operations and Resource Paths for BookStore Methods

The BookStore database holds sample book metadata, reviewer comments and inventory of individual stores. In its original state, the BookStore

offers a set of public methods that allow for local querying and manipulation of the BookStore data. Adding a REST interface is a representative engineering activity because it allows clients to consult or modify the database remotely. I will explain the sample REST API layout, and how it relates to the existing BookStore functionality in the next Chapter 6.

Note that a proper definition of REST interfaces is subject to strict design rules, notably concerning resource arrangement and naming. Those were initially described by the paradigm creator [Fie00], but have also been condensed into dedicated design rulebooks [Mas11]. An essential thought is the classification into different resource types, notably collections, variables and static resources. Some tools use visual symbolic conventions to indicate the type, e.g., variables are indicated by enclosing curly brackets, collections are annotated with a circle symbol in 5.1.

5.1.1 Relevance and Success Factors

Over the last decade, RESTful service interfaces gained widespread acceptance for modern web architectures and component-based systems, notably in a Micro-Service context [vKES+18]. This is mainly due to the efficient abstraction from implementation details, but also due to the versatility of HTTP, which provides a free choice of implementation language for communicating software components. Yet the design of a proper REST interface for a given functionality remains a challenging task. For one, correct interface engineering is subject to a variety of design rules. Secondly, the underlying web technology that enables the execution of a RESTful service imposes a complex technological stack. A side effect of this complexity is that real-world services often showcase misuse or even anti-patterns to the REST style [FTE+17].

5.1.2 REST DSLs

There exist already several DSLs that allow the specification of REST interfaces.

The nowadays most widespread interface specification language for HTTP services is OpenAPI/Swagger (OAS) [Sma23]. However, OAS is notably an umbrella language for all HTTP service interface descriptions and is not explicitly tied to the REST paradigm. This means the language does in no way enforce or foster the design of REST-adherent interfaces.

The *Web Application Description Language* (WADL)[1] was designed to describe HTTP resource behaviour in a machine-readable manner. It has been shown that WADL can be used to express the contractual interfaces of REST service implementations [FS15]. However, since WADL does not require the use of the base concepts of the REST style, it does by itself not assist or guarantee REST-compliant interface design [RR08].

The *Web Resource Modeling Language* (WRML) [Mas23] proposed by *Masse* in [Mas11] is a REST-specific modelling approach based around resources. In contrast to WADL, it therefore partially enforces the REST style. To the best of my knowledge, WRML is currently the only REST DSL proposal that considers a graphical editor that implicitly organizes resources into a tree structure 5.1.

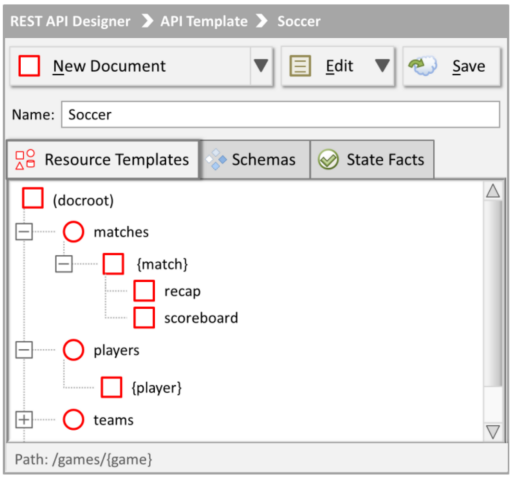

Figure 5.1: Graphical Editor for WRML's Hierarchical Resource Layout, as Proposed in Masse's REST API Design Rulebook [Mas11]

The *RESTful API Modeling Language* (RAML) is a textual modelling language that adheres closely to the REST principles [Mul21]. Interface specifications provided in RAML can be converted into OAS specifications, and from there into a variety of server and client-side stub implementations. A fundamental difference to the language provided by our concern (presented

[1]WADL is not to be confused with the *Web Service Description Language* (WSDL), designed for *Simple Object Access Protocol* (SOAP) and *Remote Procedure Call* (RPC) specifications.

in the next chapter) is that the inherent tree structure of a REST interface is not prominent in RAML specifications. Furthermore, with OAS-based code generation, it is not possible to generate meaningful working services, only service stubs. In particular, exposing already existing legacy functionality is not trivial, because the generated stubs must still integrate with existing functionality.

In summary, there are already several DSLs that allow the specification of REST interfaces. These languages focus on different aspects of the targeted services, from a technical HTTP-oriented approach (OAS) to REST-specific models with graphical support (WRML).

5.1.3 REST Annotations and Technologies

There are various ways to create a RESTful service. REST as a style only refers to the arrangement of resources and operations on them. Using Java it is perfectly possible to use JDK-provided network functionality to create a program that listens to inbound HTTP calls, analyzes their target resource and then decides how to best reply to adhere to the REST style. Technically this would already represent a RESTful service. However, this would constitute a severe mingling of technical aspects with conceptual design questions. Most REST implementations therefore draw a strict separation between the component dealing with HTTP network communication, and the resolving of resource operations and functions. In more detail, the separation is made between an application server and the RESTful service. This separation is fully in line with SoC and well established in the java eco-system, which is why there is a dedicated artifact format for web services. Where standard desktop applications can be assembled to a JAR file (Java Archive), web applications, and hence also REST services can be assembled to a WAR (Web Archive) file, for subsequent deployment on an application server. Note that there are multiple web server options, but in their functioning, they are all alike. Whether it is a Tomcat, Jetty, Grizzly or Glassfish web server, at the end of the day they all serve as platforms for hosting standardized WAR web applications.

A consequence of this separation is that the exposure of functionality over REST becomes a pure configuration task. That is, if there is already a web server, capable of dealing with inbound HTTP requests and their answering, the only missing piece is a configuration that advises the web server on how

to map inbound HTTP requests on existing Java functions. REST is exclusively about this configuration aspect.
There are different ways to establish these mapping instructions, where a web server is instructed to invoke a specific Java method in the web application, for access on a given REST resource. One of the earliest means to achieve this in the Java ecosystem is the aforementioned WADL files, which are XML files. These files can not only describe resource structure but also mapping on target functions [Bur09].
An orthogonal approach that has gained huge popularity over the last decade, and nowadays has become the preferred method, is the integration of mapping instructions via annotations. Annotations are short strings, starting with an `"@"` character, that can be placed before certain program element definitions, namely Classes, Methods, Parameters and Packages. Annotations are in general used for reflection, that is, they are interpreted at runtime, to modify program behaviour. In the context of REST, this means annotations can be used to substitute textual web-server configuration files. More precisely, we can annotate Java methods with mapping details for REST, such as a method to be invoked whenever a certain web resource is accessed by HTTP GET. Similarly we can map details of the HTTP request on method inputs. This is covered in more detail in the next chapter, alongside code examples.
Note that the annotations alone are only instructions, the configuration is only achieved by some component that translates them into web-server configurations, using reflection.

There is a subtle distinction between the annotation syntax, and the technology consuming the annotations via reflection. Regarding syntax, there are only two big players: Spring and JAX-RS. The annotation syntax is close to equivalent. For example, the annotation to expose a Java method over HTTP GET in Spring is "`@GetMapping("path/to/resource")`", whereas the JAX-RS equivalent is a combination of two annotations: "`@GET`" and "`@Path("path/to/resource")`". Concerning the technology, however, there are multiple choices for the JAX-RS syntax. That is, multiple libraries readily consume annotations in JAX-RS syntax and translate them into web server configurations, namely: Eclipse Jersey, Apache CXF, JBoss RESTEasy [Hat23, Fou21a, Fou21b].

Finally, one reason for Spring's popularity is the default integration of a preconfigured web-server [Wal22] (also called application-server). Setup and

maintenance of a web server is usually a time-consuming and error-prone task that requires proficiency with the involved network stack. However, without an application server, there simply is no platform to deploy the created web service. The success of Spring is in part due to the decision to compile by default to a JAR with a built-in web server, rather than a WAR that requires subsequent deployment. This significantly lowered the barrier to service development for less experienced developers.

5.1.4 Code Generators

There are various tools for generating server-side and client-side REST API code. The most prominent players altogether set on OAS as a specification language.

PostMan [Pos23a] and the *Advanced REST Client* [Mul23] (ARC) are visual tools for fast definition, testing, and client code generation of HTTP services. Internally the API model is stored as an OAS model and can be likewise exported. The standard use case is the one-by-one definition of sample endpoint requests, specifying resource location, HTTP method, header and body parameters. Notably, both suites allow client code generation for a plethora of programming languages.

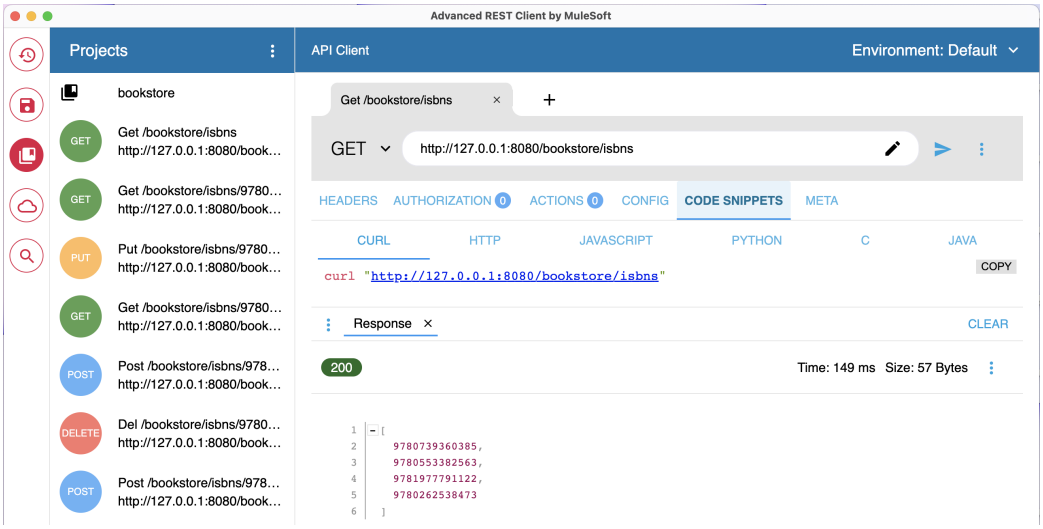

Figure 5.2: Capture of Request Specification, Code Generation, and Probing using the *Advanced REST Client*'s User Interface

OAS comes with a default API specification and backend code generation tool called *Swagger Codegen* [Swa23]. The tool generates deployable Spring Boot projects that come with a readily usable maven project build configuration. The generated REST services accept requests on all specified resource access, and answer with sample replies.

An important closing comment on existing code generators is that all aforementioned technologies only produce stub code. That is, the output is a code skeleton implementation, which contains no contextual client or server business logic. As such, the generated code must be manually edited to integrate with the application's business logic implementation. In the case of an evolving interface generation, this often implies a tedious and error-prone process, because REST-related code must be re-generated, which easily overrides previous manual changes.

5.2 Delegating Resource Access

The key paradigm behind authorized access is the notion of resource ownership. In the most common case, the owner is a platform end user, i.e. a legal person as assumed owner of a set of resources. By default, there is only one owner with operative access to the owned resources. Authorized access does, however, also consider scenarios, in which the owner desires delegation to other agents, who then act on their behalf. In that case, this third party requires formal authorization from the notional resource owner. The common protocol to achieve this securely for RESTful services (which is the most widespread cloud API paradigm) is the OAuth2 security protocol. At heart, the motivation for this protocol is to eradicate any need for credential sharing between the resource owner and third party.

A common illustration is the case of a car (resource) owned by a person (user, resource owner) [JR17]. The car is a protected resource, access and operating of which is subject to security measures (car key). In the simplest case, borrowing the car from a third party means passing on all authorizations (car keys) to the third party. The third-party can henceforth act on behalf of the resource owner. However, there is no efficient means to restrict or revoke this authorization. It is a model entirely based on trust. In the spirit of the OAuth2 protocol, the answer to this scenario is a valet car key [JR17], that is to say, a means of delegated access that comes with restrictions (speed limit, no access to trunk), and fixed expiry (borrowed car key only works for a day, then becomes useless).

It is important to note that ownership, in the context of access delegation, is a paradigm. While the application context must showcase a relation that can be interpreted as ownership, outside the OAuth2 context this association is usually not semantically equivalent to "giving the right to delegate access". Also, access delegation must be built on top of some form of authentication or access control. Coherent to this, we will see shortly that OAuth2 always includes a service component, specifically for user authentication.

5.2.1 Relevance and Success Factors

Nowadays, almost every established online platform features a notion of users and resource ownership, including the option of access delegation to third-party services. This spans from the integration of Instagram posts in dating app profiles [Bum23] to authorizing automated bank deposits for tax return purposes [CRA23], to authorizing external CI services for automated build pipeline execution on code commits [Git23, Cir23].

The OAuth2 protocol gained its status as a standard access delegation mechanism with the rise of Micro-Service Architectures [AZKA11]. The idea of service interplay lies at the heart of MSA implementations, and hence most of the well-established services would be unthinkable without a secure way of access delegation. Following the MSA context, the OAuth2 terminology is as follows: The protected resource is governed by a **Resource Server** (RS), which applies strict rules to decide on granting or denying operation access. By default, an end user holds notional ownership, this entity is therefore called the **Resource Owner** (RO). Finally, the protected resource is accessed by a third-party service, acting on behalf of the Resource Owner. The party that accesses the Resource Server is called the **Client**. Note that the client is not the Resource Owner's web browser, but a third-party service. To avoid password sharing, the authorization process includes communication with a dedicated entity, acting as a security broker that issues tokens to the Client, after successful authorization by a Resource Owner. This entity is likewise a deployed service and called the **Authorization Server** (AS) [For12].

Figure 5.3 depicts an illustration of the low-level protocol flow, based on the official specification. It has been modified to represent the request-reply nature of the underlying HTTP requests, as HTTP is the standard protocol in an MSA / RESTful context [Sch23d]. Note that the main motivation for the protocol is to achieve authorization without credential sharing.

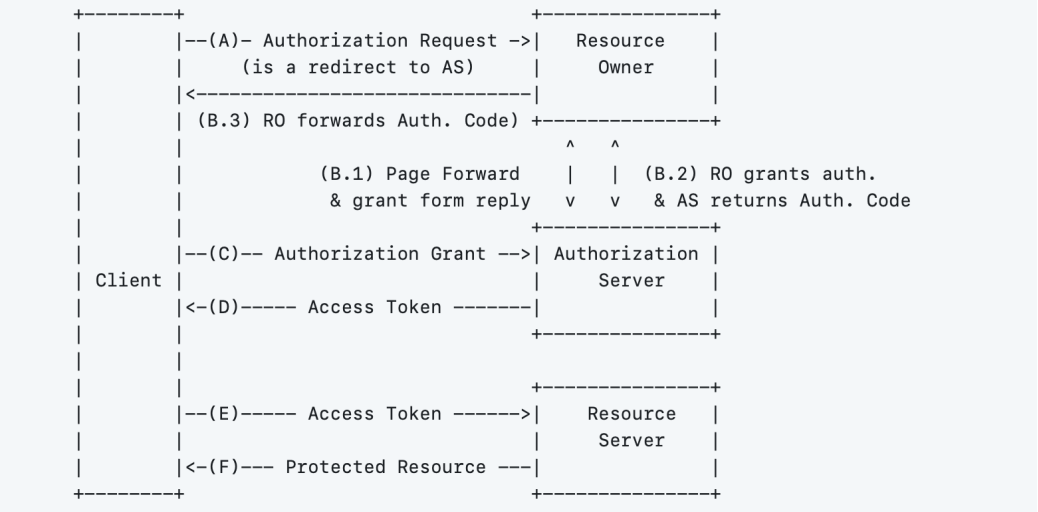

Figure 5.3: Augmented Version of the Official Default OAuth2 Protocol Control Flow [For12]

Therefore the main control flow is as follows:

- `(A)`: After an initial rejected attempt to directly access a protected RS resource, the Client (third party service) requests authorization from the RO.
- `(B)`: The RO does not simply return their credentials, but uses its credentials to create a temporary Authorization Code at the AS, which is persisted, and communicated back to the Client.
- `(C/D)`: The Client then contacts the AS to trade the Authorization Code for an Access Token. This Access Token contains the RO's identifier, as well as the privileges that were granted.
- `(E/F)`: Using this Access Token, the Client can henceforth access the RS, on behalf of the RO and interact with Protected Resources, owned by the RO.

Note that OAuth2's token-based delegated access authorization, although often sharing some implementation details with other security mechanisms, is not to be confused with e.g. single-sign-on with a platform ID, or user-to-user resource sharing. Also, integration of OAuth2 access protection does

not necessarily span out to all service resources, or in case of successful authorization grant access to all owned resources More fine-grained access is possible with a concept called "*Scopes*", which I will further detail in the remainder of this chapter.

For illustration purposes, I will now briefly illustrate the standard protocol flow, on the example of the Obscurify service [Obs23]. Obscurify accesses the Spotify music streaming API on behalf of a Spotify user and then creates statistical reports on the user's taste in music. That is, Obscurify accesses the user-owned resources for playlists, liked songs and play counts, and compares this information to other users of the same region or country. Figure 5.4 depicts the positioning of my all-time taste obscurify positioning, relative to the collected Canadian reference distribution.

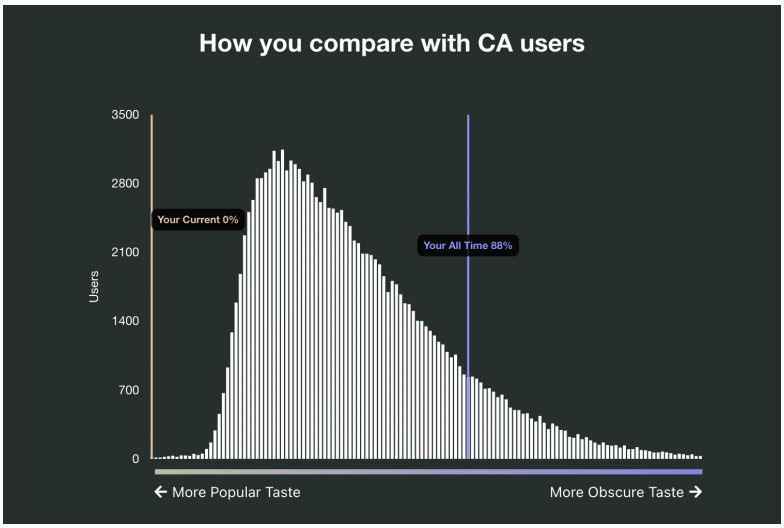

Figure 5.4: A Spotify User's Relative Taste Ranking, Compared to the Canadian Profile Distribution.

This is possible because Spotify's API adheres to the OAuth2 protocol, i.e. the API holds a notion of ownership for certain resources, and foresees authorized access on behalf of a user, without the need for password sharing. An important characteristic is that the Spotify API is agnostic to the nature of the external service, most likely even not knowing about their philosophy or purposes at the time of API definition. The Spotify developers only provided documentation [Spo23] of the authorization control flow, which in turn allows

third-party services to integrate with their API, in the spirit of an MSA interplay.

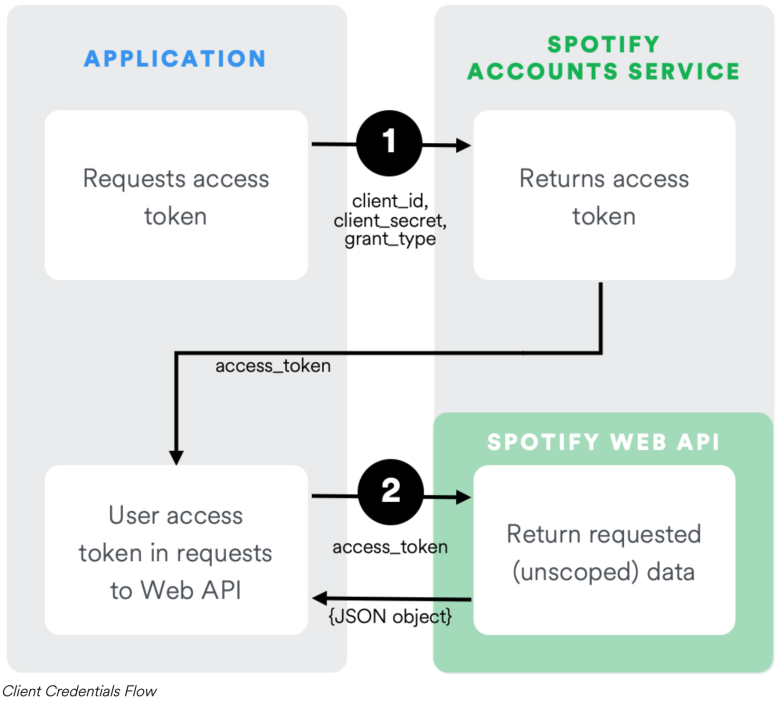

Figure 5.5: Official Spotify API Illustration for Requesting Authorized access. "Application" Represents a Third-Party Service (OAuth2 Client), such as Obscurify, Requesting Permission to Act on Behalf of a Spotify User (RO)

Note that the control flow defined by Spotify is perfectly compliant with the OAuth2 protocol specification [For12], which means that at no point the user reveals their credentials to the third-party Obscurify service.[2]. The initial informed and consented authorization request (steps `(A)/(B)` in Figure 5.3) is not contained in the Spotify API illustration Figure 5.5.

In this initial step, the user is prompted with the effects of the authorization. In the case of Obscurify this corresponds to requesting access to the user's playlists, play counts, liked songs, etc. An integral part of the protocol is the grouping of resource operation access into so-called "*scopes*". A service acting on behalf of a user in general is not granted access to *all* user-owned

[2]Request `(1)` in Figure 5.5 corresponds to steps `(A-C)` in the formal protocol specification. Request `(2)` corresponds to step `(E)`

resource operations, but only to the ones covered by the granted scopes. In other words: "*[...] scopes represent a subset of access rights, tied to a specific authorization delegation*" [JR17].
The requested authorization sample in Figure 5.6, for instance, requests three scopes, each one comprising access to a set of resource operations. Note that the provision of predefined scopes is the responsibility of the RS developer. A third-party service can only request and obtain grating of a combination of existing scopes, not define new ones [JR17].

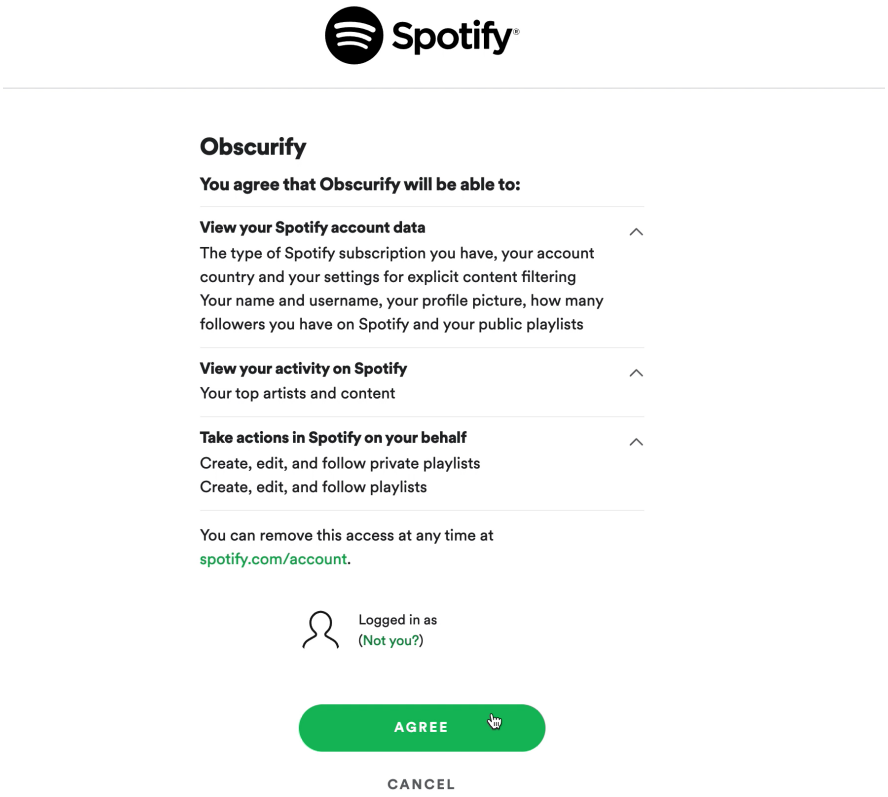

Figure 5.6: Consented Informing of the Requested Privileges, when Acting on Behalf of the Spotify User

Another key aspect of OAuth2's success is the design for compatibility with REST and therefore implicitly HTTP. As the security protocol was purposefully crafted for an MSA context, most implementations perfectly integrate with the REST paradigm and HTTP.

5.2.2 OAuth2 DSLs

Alike Spotify, most established service providers offer illustrations of their intended API usage and the designated authorization message control flow. These illustrations are often visual and model-oriented. Furthermore, the official protocol definition contains flowchart illustrations for various protocol variants. However, these illustrations are altogether neither based on a formal model definition, nor use a common, universal illustration syntax. There appears to be no generally accepted modelling notation, or even a domain-specific language, specifically created for OAuth2. Furthermore, most illustrations focus on the protocol flow, i.e. the message exchange at protocol execution time, rather than involved entities, such as agents, resources, operations and notional ownership. Nonetheless, there are several related modelling approaches to consider:

SecPAL [BFG10] is a DSL modelling approach for general decentralized authorizations. The language was proposed before the emergence of MSA and is notably not tailored to the OAuth2 context. It sets on formal definition of transitive permissions for resource access. The proposal suggested formal processing by datalog or prolog rule checkers. However, the language has not found industrial acceptance.

Several languages have been suggested for Role Based Access Control (RBAC) contexts, which are related. RBAC does not reflect the notion of access delegation or per-user resource ownership. However, it does consider the definition of privilege hierarchies as base criteria for granting or denying resource access. Microsoft's legalease [SGD+14] language provides a modelling context for these concepts. However, the language does not foresee the possibility of role or privilege multi-inheritance, which motivated the HAPI extension [JHL+21], a modified version to provide this missing feature. Figure 5.7 depicts an illustration of key concepts in HAPI, which are: (a) textual definition of Actions, Resources and Actors, and visualizations for (b) hierarchically arranged resource access operations, (c) hierarchically arranged resources, (d) hierarchically arranged user roles including multiple inheritance.
However, HAPI has likewise not found adaptation in industry. Yet HAPI is an interesting study candidate in the context of OAuth2, for it showcases a strong orientation on hierarchically arranged resources and hierarchical orders of access rights, and user roles with optional multiple-inheritance (Fig-

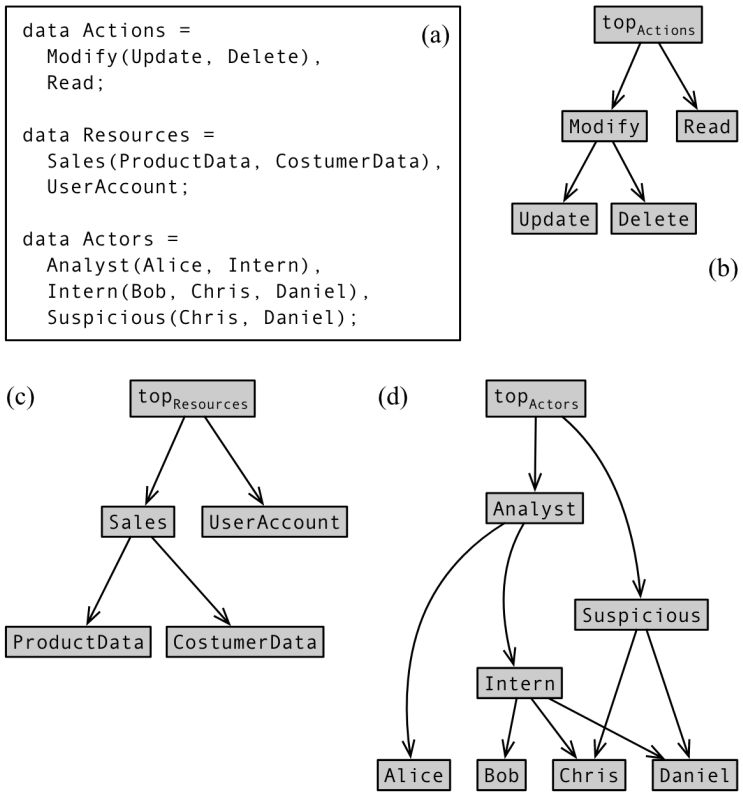

Figure 5.7: Visualized Language Model of HAPI Concepts [JHL+21]

ure 5.7), which are all language features required for representing an OAuth2 context, with a dedicated DSL.

5.2.3 Code Samples

Due to the lack of DSLs for OAuth2, there are also no code generators for interacting parties, not even for the generation of off-the-shelf stub code snippets. In the context of OAuth2, most code generators serve for the generation of test tokens, that is JSON files that comply with a given data structure of authorizations, optionally with the requested encoding and cryptographic protections [Okt23]. Figure 5.8 shows a screen capture of an online service to easily decode and encode such tokens, next illustrations of token fragmentation into header, payload and cryptographic signature.

The exact token format used is a technical variation point that has implicit consequences on the control flow. For instance, tokens can contain the RO's signature, which means the RS does not need to verify validity at the moment of Client access. Unsigned tokens, however, need an extra validation step on every use. Otherwise, token revocation would not be possible. That is, the protocol flow changes depending on which token characteristics are configured. Depending on the security requirements, one or the other token characteristics may be mandated. For example, if token revocation is a valid scenario, signed tokens should not be used, for they are valid until expiry is surpassed (if a token expiry has been configured).

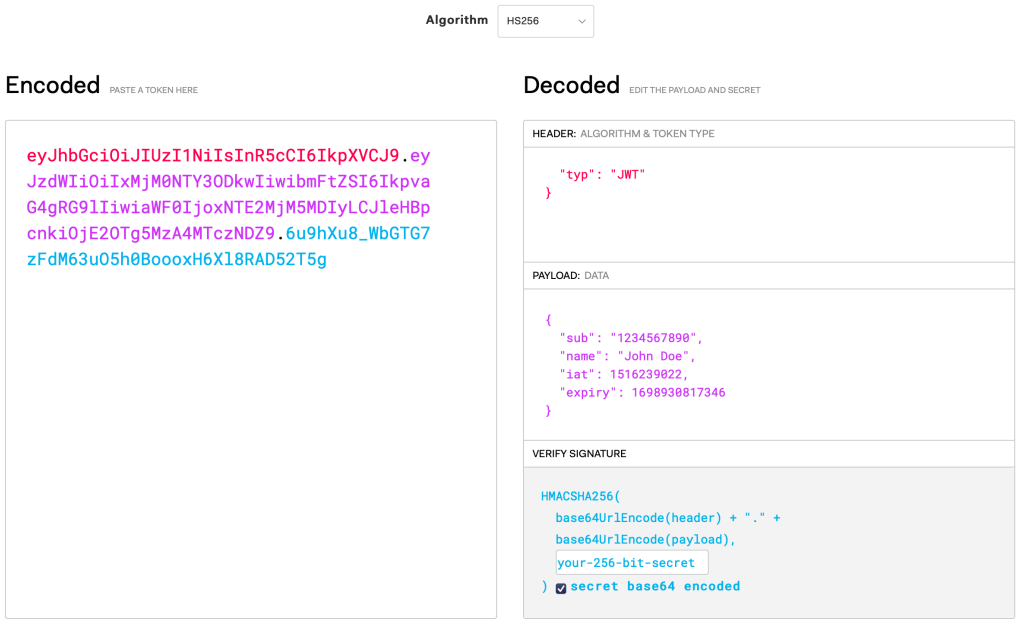

Figure 5.8: JWT.io's Online JSON Web Token Encoder/Decoder [Okt23]

Established web frameworks like Spring Boot provide mechanisms to conveniently verify token contents, by annotating existing REST resources with an additional security annotation. In the example below, a REST resource is secured by comparing a dynamic identifier in its resource path to the authorizer (RO) information contained in the access token. Note that the Spring framework here automatically extracts the requested field of the decoded token. Comparisons can be simple String comparisons, or complex functions, encoded in a dedicated DSL, the Spring Expression Language (SpEL).

```
@PreAuthorize("#city.equals(authentication.name)")
@PostMapping("/bookstore/stocklocations/{stocklocation}/{isbn}")
public void setStock(@PathVariable("stocklocation") String city,
    ↪ @PathVariable("isbn") Long isbn,@RequestBody Integer
    ↪ amount) {
     GlobalStockImpl.getInstance().setStock(city, isbn,
        ↪ amount);
}
```

Listing 5.1: Spring Annotated, Secured BookStore REST Operation. Allows Modification of Copies in Stock, if the Token's Authorizer String Matches the Store Location (city, Corresponding to {stocklocation} Path Variable)"

6

RESTify

In Chapter 5, I presented the essential background for the REST paradigm and provided references for its unmatched cloud API relevance, notably in an MSA context. Like any API, RESTful interfaces are the access point to service functionality, which in turn imposes strong requirements for a proper implementation. Poorly realized APIs, on the other hand, can easily expose sensitive service functionality or data, result in performance bottlenecks, or simply break functional correctness.

Unfortunately, the REST paradigm itself is a frequent hurdle, for many developers expose existing functionality in direct Remote Procedure Call (RPC) style over HTTP, falsely believing the protocol itself would imply coherence to the REST API design paradigm [FTE+17]. A second source of complexity is the deep technological stack, comprehending various network protocol layers, and complex deployment technology, e.g. application servers and their configuration.
The resulting development challenges and frequent engineering flaws in real-world REST service engineering are an undeniable issue and have inspired me to design a CSL-enabled *RESTify* concern that addresses these issues. At the same time, the design of such a concern serves as PoC validation of *FIDDLR*.
The previously presented CSL philosophy appears to apply to the *RESTify*

concern, as the expert knowledge provisioned by a domain-specific language could allow a streamlined REST resource layout design, and the CORE-provided model-weaving could be used to efficiently integrate newly created interface models with existing legacy functionality.

Throughout this chapter, I will first recapitulate the challenges associated with the manual conversion of legacy Java code to RESTful services. Afterwards, I explore the creation and reuse of *RESTify*, a concern to re-expose existing functionality, with the help of an internal CSL. Throughout the process, I apply *FIDDLR*, to validate the framework's designated concern design stages. The chapter concludes with a numeric comparison of the developer effort required by the two RESTification methodologies based on an action metric. I conclude with a final summary of lessons learned while implementing a functional prototype of the *RESTify* concern.

In the remainder of this chapter, I will frequently illustrate aspects of the conversion process on the example of the BookStore [Sch21] (introduced in Chapter 5), a reference sample Java application.
The BookStore imitates data models and allows access on the example of a fictitious bookstore chain. This constitutes a representative case study, for e-commerce is the original and most prominent application context for RESTful services. In more detail, the targeted resource-oriented abstraction offers the following access points:
Assortment, that is, information on all books indexed in the system are accessible at the `/bookstore/isbns` collection resource, and sub-resources. This allows adding new books to the system and lookup of previously indexed book details. Furthermore, the BookStore has a small database of *user comments* for indexed books. Those are represented by sub-resources of the `/bookstore/isbns/{isbn}/comments` resource. Operations include lookup of existing user comments, as well as deletion and adding of comments. Finally, the BookStore stores information regarding the *amount of book copies* for individual store branches. Those are accessible via the `/bookstore/stocklocations/{stocklocation}` resource and corresponding sub-resources for individual books.

6.1 Manual REST Conversion Challenges

The process of manually creating a REST interface for a Java application can be abstracted into three sequential activities:

1. Preparatory selection of a REST technology. REST can in principle be implemented exclusively using JDK internals. However, in practice, developers most often use external frameworks, in a Java context notably the Spring Boot framework. Using Spring Boot requires build system configuration file changes, framework-imposed boilerplate code and several structural code changes.

2. Exposure of functionality via annotations. This is the key activity of interest, as the actual re-exposure of existing functionality as operations on REST resources take place in this phase.

3. Deployment of the compiled service. Depending on the build process this can be either direct launching or deployment on an existing application server.

6.1.1 Variation Point Illustration

Figure 6.1 illustrates this three-phased workflow. In the remainder of this section I provide more details on each step, including sample code where applicable.

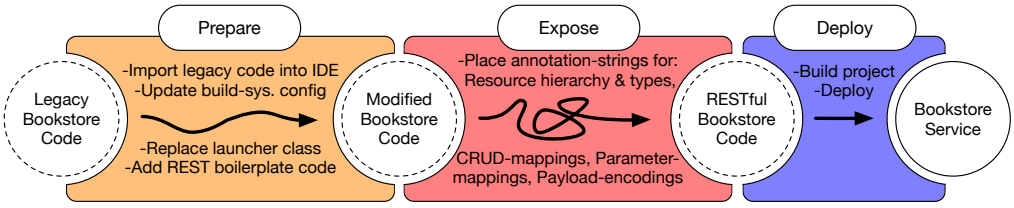

Figure 6.1: Manual Steps Required for Adding a REST Interface to Existing Code

The manual RESTification process begins with a technical choice. If a developer were to RESTify the BookStore, the first step would be the integration of a framework or library that implements the runtime communication infrastructure and protocols required for REST. Chapter 5 already listed several Java REST technologies. In the illustrations here, I assume that a developer favours *Spring Boot* over various implementations of the *Jakarta RESTful Webservices* specification (JAX-RS) [Fou21c, JBo21, Fou21b, Fou21a],

which constitute other viable alternatives. The focus on Spring is a fair choice due to the framework's high industrial relevance. However, any alternative could have been used for this study object.
Spring is a JDK-external artifact and therefore can only be invoked if referenced by the system's classpath. Thus, our developer modifies the BookStore's existing build system configuration, declaring a dependency on Spring Boot, and exchanging the default build settings by a Spring-specific plugin. Next, the developer replaces the original Java launcher class with one that initializes the Spring framework during startup. These preliminary steps are summarized in the *Prepare* stage of Fig. 6.1.

As a next step, the application API needs to be re-exposed as REST resources. Using Spring, Java methods can be mapped on CRUD operations of REST resources using Spring-specific Java annotations. An annotation parameter then specifies the resource location. An example of this syntax is shown in Listing 6.1. Spring annotations are highlighted in green. For a listing of all resource mappings for the sample BookStore application, see Table 5.1 in Chapter 5.

```
@GetMapping(value = "/bookstore/stocklocations/{stocklocation}", produces =
    "application/json; charset=utf-8")
public Map<Long, Integer> getEntireStoreStock(
    ↪ @PathVariable("stocklocation") String city) {
     return stocksPerCity.get(city).getEntireStock(); }
```

Listing 6.1: Spring Annotated BookStore Method. Accessible by HTTP GET Request, e.g. "`/bookstore/stocklocations/montreal`"

Similarly, individual parameters can be annotated where needed to resolve method arguments to details of the mapped resource query. This can be either a dynamic fragment of the resource path, an HTTP query parameter or the parsed HTTP body. Regarding the BookStore, our developer therefore identifies the existing Java methods that must be exposed and decorates their signatures with the required Spring annotations. This step is represented by the *Expose* stage in Fig. 6.1.

The modified code only becomes of practical use for remote clients, if built and deployed on a server. Building is uncomplicated, as the configured build system compiles the modified BookStore into a self-contained JAR file that can be executed as-is on any system with a compatible Java Runtime

Environment. Self-containment means that the JAR includes Spring and its transitive dependencies. If executed, the launcher class invokes Spring, which in turn powers up an embedded web server. Using reflection, Spring detects the added annotations and ensures that inbound HTTP queries are delegated to the decorated methods and that parameters are correctly resolved. The BookStore has hereby effectively become a RESTful service. This final step is illustrated by the last stage, *Deploy*, in Fig. 6.1.

In the *Expose* stage, the placed annotations *implicitly* encode an entire REST interface, that represents the *design of an entire resource tree* with selectively enabled CRUD operations and parameter mappings. This is illustrated in Table 5.1, showing resource locations and operations for a RESTified BookStore. The resource paths (second column) form a tree structure, which emphasizes the characteristic nature of the file-system-like abstraction of the REST paradigm.

The developer has to implicitly define this tree by placing annotations that encode individual branches of the tree using URLs on the exposed methods. In the case of the BookStore, the REST interface was expressed with only 28 annotations, which are scattered over the code base. For reference, a fully operational sample implementation of the RESTify outcome is available as Git project [Sch23a]. This conceptual mismatch – building a tree by writing URLs sprinkled over several source files – imposes a high mental load on the developer. Furthermore, the developer must have a thorough knowledge of the Spring annotation syntax. Hence the manual *Expose* stage is not at all straightforward and is therefore illustrated as a twirly arrow in Fig. 6.1. Only the "*build and deployment*" phase of the refactored code base are straightforward and illustrated as a straight arrow in the *Deploy* stage.

6.1.2 Source Lines Of Code Changes

An interesting observation about the described code conversion is that the development complexity does not stem from the amount of lines changed, but from the annotation syntax and the fact that annotations are scattered over the codebase. In the following, I present three tables to illustrate the issue in more detail, using the BookStore sample application, and a conversion to REST with Spring Boot.

When measuring how many *Source Lines Of Code* (SLOC) must be modified by a developer to convert the BookStore to a RESTful service, the overall number is relatively low, and likewise, only a low fraction of the code

has been touched. Yet, these modifications require significant expertise. In pure SLOC numbers, most changed lines affect the *Prepare* stage in Fig. 6.1. When a developer chooses a REST framework technology, boilerplate modifications are required that are barely application-specific. Yet this task is not straightforward and requires detailed framework knowledge (curved arrow in red box, Fig. 6.1).

Annotation	Amount
Parameter-Mapping	17
Resource CRUD Mapping	12
Boilerplate	4

Table 6.1: Annotations Added to BookStore

We can argue that the essence of the RESTification process is the correct placement of Spring annotations, sprinkled all over the codebase. Once more, the factual amount of annotations is low, as shown in Table 6.1. What renders the process challenging is identifying the target lines and the correct use of the annotation payload syntax. Concerning the last point, there is an additional challenge: When specifying resource paths, the annotations list the absolute path (unless the same root path is shared across all annotations, which then can be summarized by the Rest-Controller itself).

Resource	String replications
`isbn` (`isbns` subresource)	13
`bookstore`	12
`isbns`	8
`stocklocation`	6
`comments`	5
`commentid`	4
`stocklocations`	4
`isbn` (`stocklocation` subresource)	4

Table 6.2: Resources String Replications across Annotations

Replication is an issue, because on top of an implicit and scattered REST interface design process, the described procedure is prone to errors, simply due to the misspelling of frequently repeated fragment strings. Already on the relatively small example of the BookStore, we observe how path elements

closer to the root are replicated as Strings in annotation parameters of lower-level resource mappings. Additionally, parameter mappings may refer to resource path elements and therefore further increase String replication.

Table 6.2 shows the string replication counters for the manually converted BookStore. Unfortunately, inconsistent spelling of path segments is a real-world issue - annotation payloads are string-encoded and exempt from consistency-verification at compile time. As a result, typographic mismatches will not be detected unless the service is deployed and (hopefully) tested.

To summarize, even adding a REST interface to simple applications is subject to a tedious introduction of boilerplate code and requires sophisticated knowledge of the applied technologies. The process involves implicit design choices, scattered over the code base. We argue that existing GPLs cannot accurately capture the essence of the above design choices, i.e., the selection of a REST framework, the design of a tree-shaped resource layout and the mapping of CRUD methods and parameters on existing functionality. In the next section, we demonstrate how the above challenges can be addressed with a concern built according to the *FIDDLR* approach.

6.2 Designing the *RESTify* Concern

The purpose of the *RESTify* concern is to maximally streamline the process of adding a REST interface to expose application functionality. When applied, the concern must guide the user through the essential design choices, hide implementation details and automate any repetitive development tasks. Designing and implementing the *RESTify* concern itself, however, is not straightforward. This is where the guidance of *FIDDLR* helps. In this section, I provide a detailed overview of how the *RESTify* concern was integrated with MDE and CORE technology, following the *FIDDLR* guidelines. The illustrations below are scoped on the Spring framework, which is only one technical possibility. I later point out variation points to the process, when other technologies are used. The main concern integration steps are likewise reflected in the structure of this section, that is: The definition of concern Variants and Design Models (6.2.1), CSL Definition (6.2.2), and finally Model and Mapping Transformer definitions (6.2.3). Concerning the *FIDDLR* framework definition, these translate to the respective steps 1, 2, and 3 in Figure 4.1.

6.2.1 RESTify Variants

In the first step, the concern designer has to decide on the REST technologies the concern shall support (step 1 in Figure 4.1). Considering the implementation details seen in the previous section, the associated concern design models change at two levels:

1. **Build Configuration Template**: Depending on the technology used, different dependency statements must be included in the maven configuration. For instance, for the Spring framework, the `pom.xml` (which is an xml and therefore a tree-structured model) must contain an additional dependency node referring to Spring as compile time dependency. Listing 6.2 shows an excerpt of the design model equivalent, stating the build system dependency toward Spring.

2. **Annotation Syntax**: Depending on the REST technology, different RESTful Java code must be generated at the end of the day. For instance, while all JAX-RS derivatives share a common annotation syntax, Spring comes with its own notation. That means the Spring concern variant requires other concern-internal model transformations than its JAX-RS counterparts.

From the above, we can see that the concern variants will require different model transformations to be implemented by the concern designer.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.3.0.RELEASE</version>
</dependency>
```

Listing 6.2: Maven Dependency Statement for Spring Boot

Note that the reference implementation I implemented supports four different REST technologies. That is, all concern variants are fully operational. This is illustrated in more detail in the later section on concern reuse.

6.2.2 CSL Definition

The second step for a concern designer is to provide a custom language, tailored to the RESTification decision-making, that is, a language that allows targeted expression of the REST interface semantic (step 2 in Figure 4.1).

In the manual conversion illustration, I illustrated how the REST interface structure is implicitly defined as a consequence of REST annotations placed in the codebase. The purpose of the CSL definition is now to make the interface design an explicit process. That is, the concern designer needs to define a language that allows targeted description of the REST interface tree structure, solely by using the novel CSL and connecting it to existing legacy functionality. As a reminder, the reason why we are using a novel language for this purpose is that existing GPLs are not made for modelling resource trees. For our concern reference implementation, we therefore elaborated the *Resource Tree Language* (*ResTL*), a CSL designed for the specification of hierarchically arranged resources and basic CRUD operations. An illustration of how the ResTL language is used by the concern user follows in the next Section, Figure 6.6. For now, we are only interested in the language definition. Note that the structure of a REST API is entirely independent of the concern variant selected. This is a supportive argument for the SoC concern power of *FIDDLR*- the decision-making for API design is detached from decision-making on orthogonal technical intricacies.

Figure 6.2 depicts the CSL meta-model for the *ResTL* language. Note that the language is less expressive, and hence also smaller than existing REST DSLs. This is because the language has been designed to streamline RESTification. It essentially provides the language concepts to express the tree paradigm not available in other GPLs, thus allowing the user to focus maximally on the design of the resource tree. This can be also interpreted as: the purpose of the *ResTL* is RESTification, that is *ResTL* models gain expressiveness when placed in context with existing, classic API structures, whereas traditional REST DSLs are built for use in isolation. The main language concept to point out is the hierarchical *PathFragment* concept, which allows the user to model a tree structure. The inheritance hierarchy under *PathFragment* allows for the distinction of static path fragments (segments in classic REST jargon) and dynamic fragments. An example of the latter would be the `{stocklocation}` segment, earlier in this chapter, which is a path variable. A second noteworthy concept in the meta-model is the parameter inheritance hierarchy, which accurately reflects the main possibilities to pass parameters in an HTTP request: *Body* payload, *HeaderParameter*, *RequestParameter* and *PathVariable* (represented by a DynamicFragment). Finally, the meta-model foresees the option to access resources via the four standard CRUD access *MethodTypes* intended by the resource-oriented REST paradigm: *Get*, *Put Post* and *Delete*.

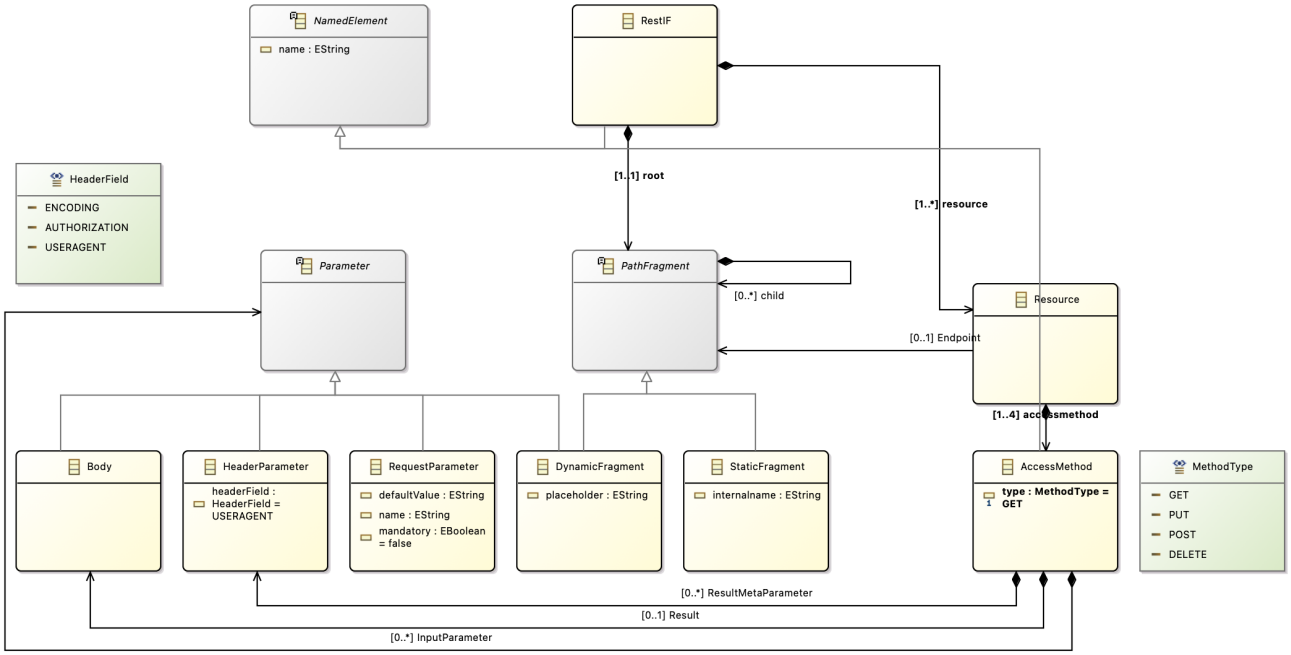

Figure 6.2: CSL Meta-Model for the *RESTify* Concern

6.2.3 ResTL to Design Model Mappings

Since the purpose of the CSL is to wrap around existing API structures (we are building a RESTify-for-legacy-software concern, not a REST-service-out-of-thin-air concern), the concern designer next has to specify how the CSL concepts align with the existing GPL API definitions of the application. In the spirit of *FIDDLR*, this is done using *Language Element Mappings* (LEMs). Note that the concern designer does not need to define a novel language just for defining the mappings. With LEMs, CORE already provides a generic artifact for 1:1 model element mappings, which allows the concern user to map CRUD operations to functional elements of the base application, i.e., the methods our sample BookStore offers. An illustration of how the mappings manifest in a graphical editor is depicted in the subsequent section on concern reuse, Figure 6.6. For now, we are only interested in defining which concept mappings must be considered.

In principle, *FIDDLR* does not enforce LEM definitions to occur at a specific level of abstraction. Regarding the framework, this decision was made with the maximized reuse of existing MDE and CORE concepts in mind. The concern designer can autonomously decide at which MDE-level of abstraction a novel CSL is best integrated with existing GPL models and code of the base application. For *RESTify* we decided to perform the integration at the design level only, e.g., using class diagrams and sequence diagrams, and rely on standard MDE code generation to produce the running application.

For *RESTify*, the LEMs allowed by the concern for mapping the novel *ResTL* language on existing contextual GPL models are as follows:

1. *ResTL Access Methods* can be mapped to GPL Object method signa-

tures. That is, *Get*, *Put*, *Post*, or *Delete* operations on a REST resource can be associated with method signatures of classes.

2. *ResTL Dynamic Fragments* can be mapped on GPL Object method signature parameters. That is, a path variable placeholder can be associated with a primitive input parameter to an existing method signature function. Note that non-string primitives are automatically converted to the target primitive type, where possible.

Note that the concern foresees a mechanism for body payloads. If a mapped GPL class method signature showcases *exactly* one unmapped signature parameter, then it is considered as mandatory Body Payload of the associated REST operation. This notably means that mappings, and even the absence of mappings, carry more implicit semantics than in traditional CORE approaches, where mappings are simply model weaver instructions of equivalent concepts. This is explicitly not the case with CSL concerns.

Note that the concern designer may optionally need to create a graphical editor, that can express both the artifacts created by he newly defined language (resource tree defined by *ResTL*) and the mappings. However, textual model editors are common practice and not every new language may need such visual tools.

6.2.4 Transformers

Following *FIDDLR*, the concern designer's third and final activity is to provide one or several model transformations that transform the CSL model into GPL design models, i.e., that convert the mapped *ResTL* models into class diagrams and sequence diagrams. Throughout this chapter, I proceed with illustrations for the Spring framework. In this case, apart from annotations we also need to generate models to trigger the Spring launcher behaviour during the startup of the application. Other REST technologies, as specified through the Variation Interface, explicitly require slightly different transformations. I delve into this subject in more detail in Chapter 8.

Per variant, we are now interested in a language transformer, that converts models defined in the *ResTL* CSL to GPL models (step 3 in Figure 4.1). Such a transformer must likewise take the given mappings from CSL to GPL models into account (step 4 in Figure 4.1). For every REST resource operation, mapped on a GPL method signature, the transformer creates a stub GPL class with the same signature and the corresponding REST annotation. Notably the annotation syntax changes, depending on the concern variant

previously selected. In addition to that, the transformer generates a GPL sequence diagram model for the method stub. This model also describes a GPL method call, which serves as the connection point to the original, unrestified codebase. A newly generated mapping from the generated GPL model to the original GPL model serves as *composition specification*. This composition specification, when given to the CORE weaver, composes the GPLs of the base application with the generated GPL models containing the REST-specific information.
Additionally, the already described conversions, the transformer must also encode Spring annotations for mapped parameters, which can be either segment variables or body parameters, depending on the mapping context.

No further work is necessary. Notably, it is not required to implement an adapted weaver or code generator. The outcome of the described transformers is pure GPL models, which in turn means that the standard CORE weaver is used to compose the design models, and a standard MDE code generator can generate the executable. In our case, this tooling is provided by the CORE reference implementation, TouchCORE [Lab21].

In summary, from the perspective of a concern designer, when following *FIDDLR*, several things must be provided: Concern variants and associated design models, a novel *concern-specific language* and the two *model transformations*, generating the GPL models and the composition specification. Reused technologies are the mappings and the weaver provided by CORE, the code generator provided by MDE, and the Spring framework (and potentially other REST technologies) itself.

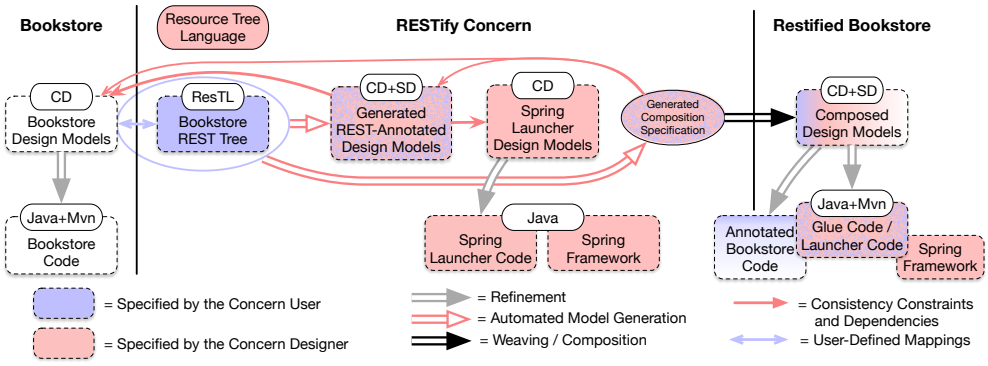

Figure 6.3: *FIDDLR* Applied to the *RESTify* Concern

In the case of *RESTify*, we can illustrate the described concern designer activities by highlighting the aforementioned activities in the overall *FIDDLR* design process diagram. This is visualized in Figure 6.3.

In summary, the concern designer activities are:

1. Provision of Variants and variant-specific Design Models: *spring launcher design models*
2. Definition of a custom concern-specific language, including LEMs: *resource-tree language*
3. Definition of CSL to GPL transformers: *automated class and sequence diagram generation*
4. Definition of LEM to composition specification transformers: *GPL model weaver instructions generation*

Table 6.3 maps these concern designer activities directly on the original framework steps, as illustrated in Figure 4.1.

FIDDLR Step	*RESTify* Concern Designer Element
Step 1: Realization Models	Spring Launcher Class, Maven Code
Step 2: CSL Definition	Provide *ResTL* Meta Model
Step 3: Model Transformers	*ResTL* to Java Annotations
Step 4: Mapping Transformers	*ResTL* Mappings to Weaver Instruction

Table 6.3: Mapping *RESTify* Design Actions on *FIDDLR*

6.3 Applying the *RESTify* Concern

So far we have seen what technical intricacies occur for a developer in a manual conversion to REST and how a concern designer can develop a CSL-enabled concern to address these challenges by following the FIDDLR framework. The purpose of this section now is to illustrate how adding a REST interface to an existing application is experienced from the perspective of a concern user, that is a developer who seeks to RESTify legacy code, by using the concern instead of pursuing a manual conversion.

In essence, the reuse process reflects the key stages of classic CORE reuse, that is, guided decision-making in the three dimensions of reuse [KMA+16]. I will now illustrate this process in more detail, on the example of the initially

presented BookStore. Throughout concern reuse, every stage is facilitated by a dedicated graphical model editor, to allow explicit, but assisted decision-making for all essential design question. Where applicable, I also highlight the differences between the model-oriented process compared to the classic, manual approach.

6.3.1 Variant Selection

Equivalent to manual conversion to REST, the process starts with selecting the desired REST technology. Where in the classic conversion a developer needs expert knowledge on viable alternatives and the technical proficiency to realize their integration, *RESTify* offers this choice through a CORE-based variation interface (VI) that captures the technologies considered by the concern designer as shown in Fig. 6.4. Note that the graphical interface for variant selection is a standard TouchCORE component and does not need to be redeveloped by the concern designer. In full compliance with the previous section, the available options are arranged hierarchically. This makes sense because various technologies internally make use of the same JAX-RS annotation syntax.

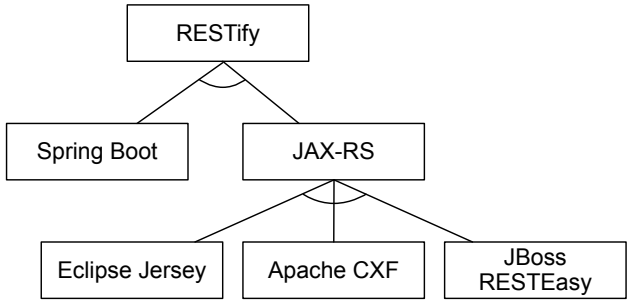

Figure 6.4: Variation Interface of the *RESTify* Concern

Variation Interfaces (VI) can also contain information on the impact of user-made choices on resulting software qualities of the outcome, such as *performance*, *security*, *etc.* This information stems from an optional goal model provided by the concern developer. In the context of REST technologies, this could for instance provide general insights of decision impacts on service security or projected throughput.

6.3.2 CSL Modelling

Once the desired technology is selected, the user is brought to the *ResTL* model editor. The concern user then models a possible resource layout as shown below in Figure 6.5, assisted by the editor that enforces a coherent layout. Thanks to the *ResTL* CSL provided by the concern designer, the concern user is maximally focused on this REST-specific design task. In the case of *RESTify*, the spotlight is on the definition and organization of resources in the form of a tree and exposing CRUD operations. Detailed REST interface information, e.g., input and return parameters, or meta-information, such as preferred HTTP payload encoding, is purposely omitted at this stage.

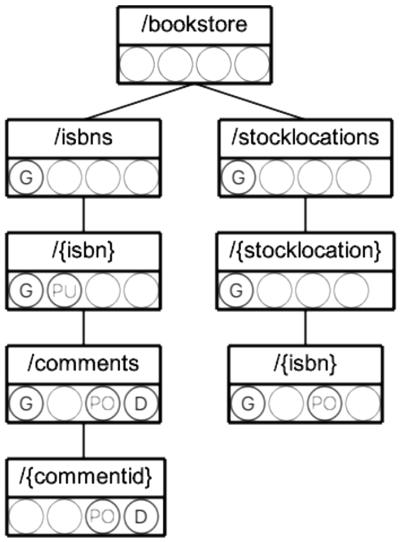

Figure 6.5: BookStore Resource Layout Designed with the *ResTL* Editor. Circled Letters Below a Resource Represent Enabled CRUD (**G**et, **Pu**t, **Po**st, **D**elete) Operations

This illustrates one of the key differences between a standard DSL and a CSL. While a REST-DSL would have to specify detailed parameter information, the *ResTL* language does not. It integrates perfectly into the concern reuse workflow and in its function as a wrapper language only provides concepts complementary to the GPL given application context.

6.3.3 Mapping to Application Context

The last step consists of connecting the newly created CSL model with the existing BookStore application logic, that is, the concern user must now establish mappings that turn the defined resource structure with CRUD operations into a wrapper for the existing BookStore application interface. Note, that it is not necessary for the concern user to manually redefine models for the BookStore application. CORE comes with a built-in automatic signature extraction from existing artifacts, e.g. JAR files. From the perspective of a concern user, the starting point is a split view with the CSL resource model to the left and the existing BookStore method interface to the right. The sole task is to create point-to-point mappings (or in easier terms: *lines*) between these two models.

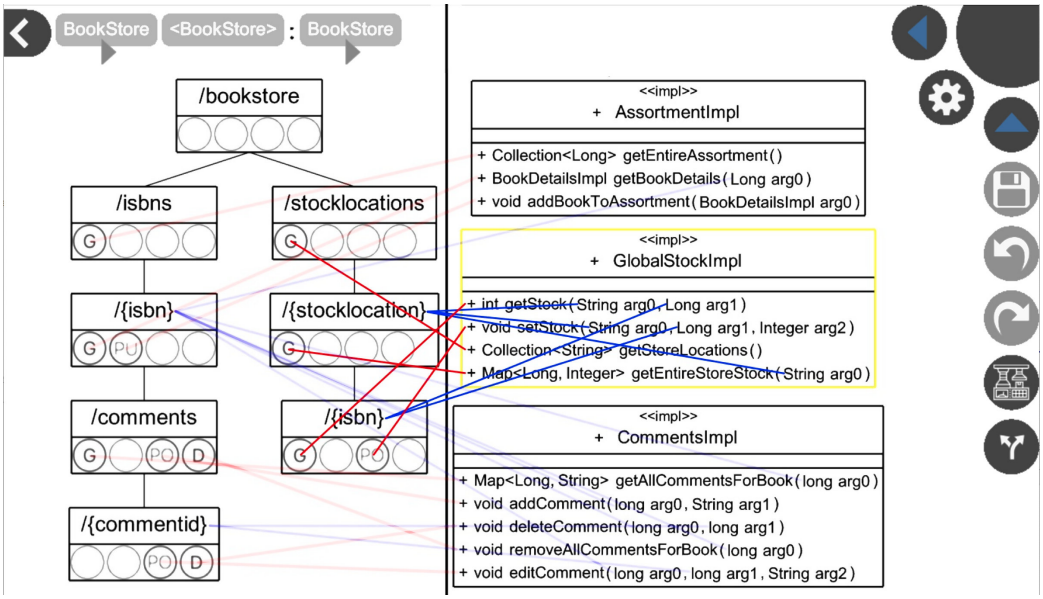

Figure 6.6: TouchCORE Screenshot Showing Split View in Action. Mappings can be Highlighted Selectively to Improve Visibility

Larger interfaces can easily result in many overlapping lines, which makes it difficult to follow individual mappings. This is why the reference implementation provides a simple mechanism to maintain a good overview. Only mappings for concepts visible in the current zoom level are displayed. By zooming in or out, mappings can be conveniently created and inspected more selectively.

Once the mappings are defined, operational code can be generated with the press of a button. The concern provided transformers automatically convert from the CSL model and mapping information to standard GPL models and weaver instructions, that are readily consumed by the existing CORE MDE pipeline. The outcome is maven / Java code, that can be compiled and deployed with minimal effort, and effectively exposes functionality as a RESTful service.

Note that in coherence to the LEM definition provided by the concern designer, *RESTify* considers two types of mappings: links between individual CRUD operations and existing BookStore methods, and mappings between signature parameters and intermediate dynamic resources. These mappings originate from dynamic path segments (denoted as a placeholder enclosed by curly brackets) and are indicated in blue in Figure 6.6.

Finally, a reminder that not all target signature parameters need to be mapped. The remaining unmapped parameters are assumed to represent body payloads. Note that in REST there are also query parameters, which are parameters passed as an appendix to the URL, separated by a question mark. Currently, this parameter type is not supported by *RESTify*.

Compared to the original, manual code conversion, we can retain that the described concern reuse process provides a substantial streamlining of decision-making. This is summarized in Figure 6.7, indicating how the concern reuse activities improve on the original conversion challenges.

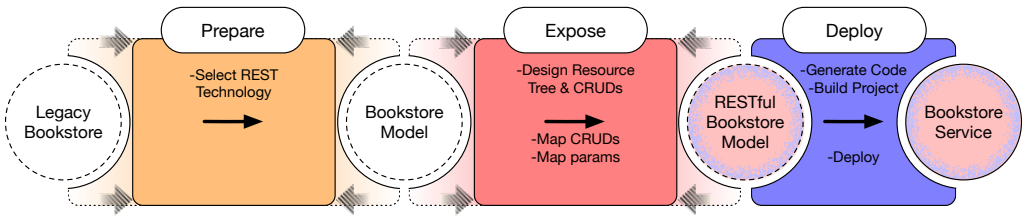

Figure 6.7: Application of the *RESTify* Concern. Preliminary Tasks are Reduced to a Minimum. Decision-Making is Explicit

It is no longer up to the developer to identify technological alternatives (and subsequently figure out the sometimes time-consuming integration at the code level). All supported technologies are specified as features in the variation interface, and can be conveniently selected by the concern user. Afterwards, the developer no longer has to sprinkle complex annotation all over the code base in a time-intensive and error-prone process but benefits from a

tailored modelling syntax to express the exact nature of a meaningful REST wrapper in the form of a mapped tree. Finally, since not only source code but also build configuration specifications are generated, service deployment remains a straightforward activity.

A final, unmatched advantage of the *RESTify* concern is the built-in possibility for belated decision-making. In the manual approach, it is not easily possible[1] to revert a decision on the REST technology, once a legacy project has been diligently converted. In the worst case, a belated technology switch not only affects boilerplate code changes and build configuration modifications but also affects the REST annotation syntax itself. If e.g. the task is to switch from Spring to a JAX-RS derivative, every REST annotation needs to be rewritten. Listing 6.3 shows the JAX-RS semantical equivalent of the previously presented Spring annotations in Listing 6.1 (Spring-Boot syntax). The technology change would require manual rewriting of all code shown in orange.

```
@Path("stocklocations")
public class GlobalStockImpl {
  [...]
  @GET
  @Path("{stocklocation}")
  @Produces("application/json")
  public Response getEntireStoreStock(
    ↪ @PathParam("stocklocation") String city){
    return Response
      .status(Response.Status.OK)
      .entity(stocksPerCity.get(city).getEntireStock())
      .build(); } }
```

Listing 6.3: JAX-RS Annotated BookStore Method. Accessible by HTTP GET Request, e.g. "`/bookstore/stocklocations/montreal`"

Using the concern, however, a technology switch can be achieved within seconds. The concern user simply loads the already finished concern models and reverts the choice made in the initial VI. Since the CSL and mappings are independent of the selected variant, new code can be generated and deployed without any further actions needed.

[1] Most developers would actually consider such a request an utter nightmare.

6.4 Qualitative and Quantitative Comparison

In this section, I briefly contrast the two approaches concerning several high-level software engineering goals. In more detail, I do so regarding Separation of Concerns, redundancy, and prospective potential for service evolution. Finally, I provide a brief action metric to quantify the efforts associated with either approach.

Regarding separation of activities, *RESTify* efficiently offsets the convoluted choices embodied in the manual approach: Adding a REST interface to an application is done in three separate steps – selecting a technology, designing the resource layout, and establishing the mappings to the application. Each step is as simple as possible, performed at the right level of abstraction supported by the right modelling notations. No expert REST knowledge is required by the concern user. The manual approach however mingles decision-making with a need for advanced technical details, e.g., framework-specific boilerplate code, annotation syntax or intricate configuration file modifications.
The main difference is that in the manual approach, the developer spends a significant fraction of their overall efforts on preliminary or boilerplate tasks and implicit decision-making (implicit design of an API layout utilizing scattered annotations). With *RESTify* the process is guided and straightforward with explicit decision-making and minimal overhead. Fig. 6.7 illustrates how the gains manifest in the three main restification stages.
Furthermore, from a modelling SoC perspective, it can be considered beneficial to have some aspects of a REST interface, i.e. the resource structure, visible at a glance in a dedicated tree model.

Redundant or duplicated configurations are a common engineering pitfall. In the case of manual refactoring to REST, the most prominent form of redundancy occurs in the resource path specifications. As previously discussed, and illustrated in Table 6.2, notably resources closer to the resource tree's root have a high likeliness for string replication, and hence also incoherent spelling. Where the manual BookStore conversion showcased this severe replication of resource strings (scattered over annotations in multiple files), the *RESTify* models define every resource name exactly once, hence eliminating this error source in the first place.

If there is one thing guaranteed in SE, it is software evolution [FRBS04]. For instance, the API of a restified service may expand or restructure, or

Element	Occurrences in BookStore	Actions
Technology select	1	1
Resources	8	24
CRUD Operations	12	12
CRUD Mappings	12	12
Parameter Mappings	13	13

Table 6.4: Atomic Actions to Convert BookStore with *RESTify*

changed design requirements can require a restructuring of the modelled REST API. In a purely code-based approach, this can be tricky, for the overall interface is not visible at a glance. *RESTify* however facilitates interface evolution, which is an effect of changes occurring at a meaningful level of abstraction. For example, restructuring the REST interface's URL tree is as simple as rearranging the resource tree layout in the *ResTL* editor. But even more complex evolution scenarios are considerably simplified. As previously mentioned, the *RESTify* concern supports belated technology switching, e.g. switching from the Spring Boot-based implementation to a JAX-RS-based implementation. Using *RESTify*, this is as simple as selecting a different feature from the variation interface of the *RESTify* concern. A manual migration from one technology to the other would constitute a considerable effort because here the annotation syntax differs between REST frameworks. In summary *RESTify* greatly improves the overall potential for service evolution, compared to the manual approach.

By nature, it is hard to define a quantitative metric for comparing a modelling approach to a code-based approach. We applied an action-based metric to estimate the efforts required for a concern-based conversion of the BookStore. That means, we defined atomic actions for all individual modelling activities and counted the number of such actions needed to add a REST interface to the BookStore.
The atomic modelling actions that need to be performed in *RESTify* are as follows: The desired technology has to be *selected* in the VI with a single click. Creation of the tree model in the provided reference editor requires three interactions for each modelled resource: one *create* instruction, one interaction to *specify* the resource type, and a third interaction to *enter* its name. Finally *Exposure* of an available CRUD operation for subsequent mapping is achieved with a single click.

In addition to the previous activities, every resource or parameter *mapping* requires an additional action. For the BookStore we end up with a total of 62 modelling actions, as shown in Table 6.4. This represents a greatly reduced effort when compared with the over 100 lines of source code that have to be written, modified or removed in the manual approach. We believe the comparison and conclusion drawn is fair, if not underselling the concern's potential. The numbers show that when comparing both approaches, there are more textual code changes required than graphical interactions. Furthermore, every textual modification can be considered more complex than any atomic action (which is just a mouse click). This first quantification therefore suggests a simplification of the RESTification process by the *RESTify* concern.

6.5 Lab Validation

On paper, the *RESTify* concern definition served as a first proof of concept for the *FIDDLR* framework. It validated the feasibility of defining a concern with integrated CSL that efficiently connects with existing CORE and MDE tools, hence maximizing tool reuse. Considering the *FIDDLR* concern design activities, the transformer definitions constituted by far the greatest effort, followed by finding a meaningful level of expressiveness for the *RESTify* language.

Furthermore, I implemented an operational reference implementation of the *RESTify* concern that supports all major four Java REST technologies. This more technical step served as validation that the language definition and transformations are purposeful and implementable.

On a side note, the *RESTify* language also proved useful as a simplified, graphical way of explaining and communicating REST interface overviews in a classroom setting. Although the language does not suffice on its own as a wholesome REST interface definition language, the students embraced the language as a helpful support when manually adding REST interfaces to student projects. Most developers perceive modelling languages and modeling-related activities mainly as sketching, or exploratory activities, not code equivalent development [DEPC21].

Furthermore, the concern reference implementation was tested with several sample applications and allowed flawless generation of read-to-deploy back-end code. In lab internal experiments concern-based software conversion was significantly faster than comparative manual activities. This is to a huge part

due to the seamless integration with the TouchCORE workbench, which gave access to a sophisticated model weaver and code generator. The integration worked as expected.

However, running a lab-internal case study is naturally biased. As a developer of the reference implementation, I am not a representative test subject for an objective evaluation. Therefore, to gain insight into the question of whether a concern implementation based on *FIDDLR* can accelerate the development of REST interfaces for applications, I designed and conducted an extensive empirical experiment with 28 participants. The details and outcome of this experiment are presented in Part III of this thesis.

7

AUTHify

In chapter 5, I've presented the main control flow of the OAuth2 access delegation protocol and listed references to existing tools and the protocol's industrial relevance. Like most security aspects, ensuring correct compliance with the protocol at the application context level is an engineering activity that requires diligent precision. Even little implementation flaws may represent a threat to system integrity and security. In the case of OAuth2, software engineers consider even now in 2023, years after the protocol's emergence, that correct integration and configuration of the OAuth2 protocol is a lasting challenge [Gul23]. This impression is consistent with continuous reports of incorrect protocol applications, even in well-established online services like *Grammarly*. In the latter case, a seemingly small configuration inconsistency affected thousands of users and exposed their private data [Car23]. On the bright side, this renders the OAuth2 protocol an interesting study object for concernification and CSL toolchain support. If accurately expressed through a CSL, security expert knowledge could be readily shaped around existing, unsecured REST APIs, and effectively mitigate the present implementation challenges. Guided mapping of security paradigms from a custom delegation language on existing API models could be an elegant way to sidestep protocol configuration pitfalls.

In this chapter, I discuss *AUTHify*, a novel CSL concern, tailored for applying the OAuth2 protocol to REST APIs, and enabling fine-grained authorization of third-party services without a need for password sharing. An important detail about the *AUTHify* concern is, that it builds on top of *RESTify*. While in general, access delegation is relevant for all RESTful services, regardless of how they were created, I was particularly interested in how CSL-concerns perform when stacked. Therefore, I decided to investigate the *AUTHify* concern as an extension to the *RESTify* concern. That means, *AUTHify* only allows the securing of services that have been previously built with *RESTify*. Other services, e.g., REST services that were manually built are not considered by the *AUTHify* concern. We do, however, provide a short discussion of means to mitigate this limitation in the future work section of Chapter 14.

Also, note that the concern is strictly limited to the access delegation aspect. Potential extension to other security concepts is discussed in Chapter 14.3. I now will first recapitulate error sources on the example of a manual OAuth2 integration approach, then demonstrate the conceptual design of *AUTHify* with *FIDDLR*, and finally demonstrate how concern usage is perceived from a reuse perspective and how this eases the protocol integration activity, compared to the manual approach. Note that while the OAuth2 protocol defines several roles, this concern only deals with the Resource Server, that is, the server providing the REST API to be protected.

Throughout the remainder of this chapter, I will illustrate technical details on the example of a modified version of the BookStore [Sch21]. In this version, we assume a part of the REST interface under the ownership of respective bookstore managers, who in turn wish to grant access to various external services (Clients) acting on their behalf. I will first present the modified BookStore API, then illustrate the definition of sample *Scopes* for three exemplary OAuth2 Clients. For completeness, I also provide functional reference implementations for several Clients [Sch23d]. The referenced code is based on an OAuth2 sample setup by Baeldung [Bae21], which has been adapted to the BookStore context, to maintain continuity throughout this thesis.

7.1 Manual Service Securing Challenges

I start by presenting a slightly modified version of the BookStore API. Figure 7.1 illustrates the changes made, compared to the *ResTL* model presented in the previous Chapter 6.

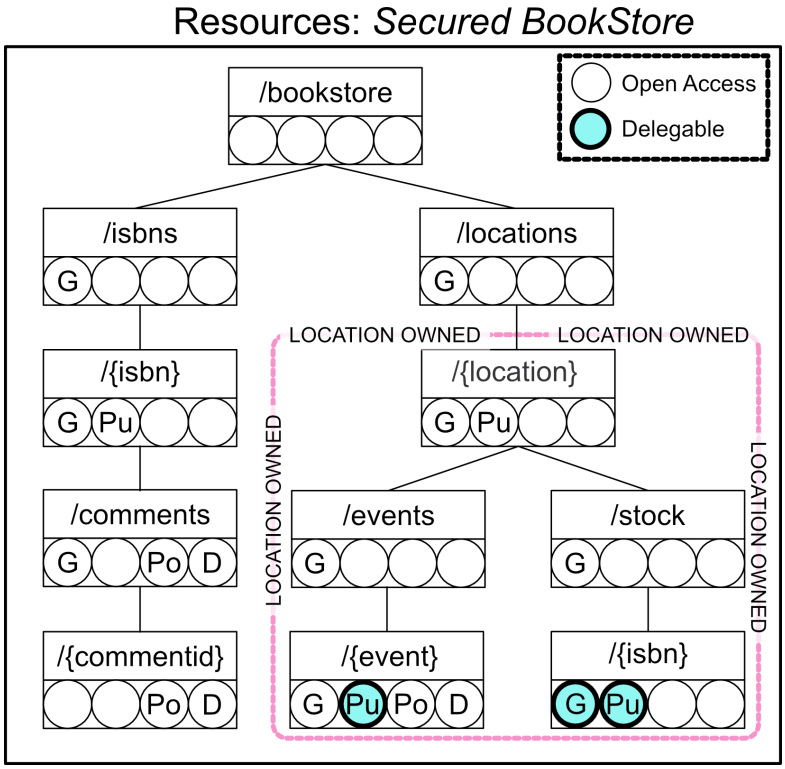

Figure 7.1: Illustration of the Modified BookStore's REST API

For now, we are only interested in the structural API changes, not yet the access restrictions or scopes (fat circles and turquoise fills). Note that only the right half of the resource tree has been modified, everything left of the root resource is unchanged.

The main change is that locations now do not exclusively describe stock information, but represent details of a specific location. The following breakdown describes the presented resource operations details, relative to the `/bookstore/locations/` sub-resource:

- `/{location}`: represents the store location. We consider this resource and everything below conceptually owned by a store manager. A public

`Get` operation enables the lookup of opening hours and address information. `Put` allows the caller to update the information.

- `/{location}/events`: represents public events taking place at a given bookstore location. `Get` returns a public list of events.
- `/{location}/events/{event}`: makes it possible to invoke CRUD operations on individual events. Customers can look up event details with a `Get` operation, and a store manager can create, update or delete individual event descriptions with respective `Post`, `Put` or `Delete` operations.
- `/{location}/stock/`: represents all books in stock. A `Get` does however only provide a list of all books currently available, not the actual number of copies.
- `/{location}/stock/{isbn}`: represents the number of book copies available in store. A book manager can look up the exact number with a `Get` operation, or change the number using a `Put` operation.

We can easily imagine how several of the above resource operations should be restricted. For instance, modifying details of local events, like book readings, should only be done by the bookstore manager. All operations that we consider protection-worthy are indicated by bold circles in Figure 7.1. However, the interest of this chapter is not classic access securing, but OAuth2 authorization scenarios. That is to say, we are interested in scenarios where external online services access the API, authorized on behalf of a given manager. Before I delve into the details of API securing, I will first present three representative external services, or OAuth2 Clients, to access the presented API on behalf of a manager:

1. **Event Checker Service**: The interest of this service is to verify the currently advertised upcoming event descriptions and ensure their textual description is free of typos. Not a very complicated service, but we can imagine how a bookstore manager has an interest in authorizing such a service to act on their behalf.
2. **Low Stock Alert Service**: From the perspective of a bookstore manager, nothing is more frustrating than a client who would like to spend some money, but the book they are looking for is out of stock. Luckily

there's a third-party service that can be configured to send an alert email when the number of remaining copies for the bestseller books are below a critical threshold.

3. **Auto Stock Replenish**: Finally, the bookstore manager regularly orders new copies for the books that ran out of stock. Of course, when some days later a truck arrives, the manager could manually check which book copies were delivered, and update the stock information accordingly. But the delivery service can also take care of that directly, on behalf of the manager, since they know the delivery details.

Whatever the service, the bookstore manager could of course simply pass their credentials to each contractor and hope they only use it for the requested purpose. The more common approach for modern service architectures is access delegation with the OAuth2 protocol. In that case, the manager can conveniently authorize each service to act on their behalf, without password sharing - and notably restrict service access to only the functionality needed for the use case in question, using access scopes.

Note that the definition of scopes falls under the RS developer's responsibilities. That is to say, whoever decides on scopes may anticipate, but is not familiar with the exact services requiring API access. For illustration purposes, I here assume that the designer happened to foresee the same services as listed above.
The following enumeration lists for each OAuth2 Client, which resource operations are minimally needed:

1. **Event Checker Service**:
   Modification access (`Put`) on `/{location}/events/{event}`

2. **Low Stock Alert Service**:
   Read access (`Get`) on `/{location}/stock/{isbn}`

3. **Auto Stock Replenish Service**:
   Read and write access (`Get` and `Put`) on `/{location}/stock/{isbn}`

With these services in mind, an RS developer could define one scope for read and write access to events, one scope for read access to stock information, and one scope for write access to stock information. Note that in this example, the **Auto Stock Replenish** Client requires a combination of two

scopes to function. According to the OAuth2 protocol definition, overlapping scopes are a technical possibility, but for the design of this concern they were intentionally restricted. This is discussed in more depth in Section 7.4.1.

Code Complexity

With the OAuth2 Client, RS and scopes defined, I now provide technical illustrations of the code modifications needed to support the aforementioned access delegation scenarios.

Regarding the RS, securing a REST endpoint for access delegation comes down to two separate security checks. Both are performed each time an OAuth2 Client (third-party service) accesses a secured API endpoint on behalf of a RO:

1. **Matching of the Resource Owner**: The authorizing entity must match the resource owner. That is when a protected operation of a specific store location is accessed, the access delegation authorizer must be the owner of that store.

2. **Verification of granted Scopes**: The OAuth2 Client must additionally hold the required scopes, that is, they must be authorized to access the requested resources. Holding scopes is required, because authorizing a Client to perform delegated access on some operations, is not equivalent to granting them access on all operations. Scopes grant access to a reduced subset of all access restricted operations.

Using Spring Security, these two checks are implemented with two separate protection mechanisms: `@PreAuthorize` annotations for ownership verification, and `FilterChains` for scope verification.

Both mechanisms set conditions that restrict access, and both must be evaluated positively, for a third-party Client to gain access. I will now illustrate the security checks on the example of the read operation, namely lookup of the exact number of copies in stock for a given book. Listing 7.1 shows how a regular Spring Boot `Get` operation on `/{location}/stock/{isbn}` is secured with `@PreAuthorize`:

```
@PreAuthorize("T(GlobalStockImpl)
    .getInstance().getStoreManager(#location)
    .equals(authentication.name)")
@GetMapping("/bookstore/locations/{location}/stock/{isbn}")
  public void getStock(@PathVariable("location") String location,
    ↪ @PathVariable("isbn") Long isbn, @RequestBody Integer
    ↪ amount) {

    return GlobalStockImpl.getInstance().getStock(location,
      ↪ isbn);
  }
```

Listing 7.1: Spring Security Annotation to Verify Delegated Access Occurrs on Behalf of the Legit Owner

The red parts in Listing 7.1 are interpreted as follows: Before invoking the REST operation, Spring Security inspects the Json Web Token (JWT) sent along with the HTTP request. This token was issued to the Client by the AS, at the moment the RO granted delegated access. The token is a JSON file, which contains an `authentication` field, the value of which is the UserId of the RO who authorized the Client to access the REST operation on their behalf. Depending on how the user management is implemented, this can be a unique name, a number, an email address, or any other unique identifier.
The interest of the `@PreAuthorize` annotation payload is now to ensure that the RO who granted access owns the requested resource. The annotation's payload is an expression, which resolves to a boolean value. Access is granted or rejected, based on the evaluation of the expression. Here the expression ``"`T(GlobalStockImpl).getInstance().getStoreManagerName(#location).equals(authentication.name)`" accesses the BookStore's internal database, to verify if the provided username is the valid owner of the requested location. The value of `#location` is given by the request's corresponding `PathVariable`.

As previously mentioned, allowing a third party to act on behalf of the owner by granting access to some operations is not equivalent to unlimited impersonation. Scopes make it possible to restrict authorized third-party clients to specific subsets of operations. A client without any granted scope cannot interact with the RS REST API, even if correctly authorized by the RO. In the most extreme case, this can be exemplified with a token that does

not list any scopes. It was obtained by legitimately following the protocol flow, but does not grant access to any protected resource.
Next to the `@PreAuthorize` annotation, Spring Security therefore provides a concept called `FilterChains`, which allows additional testing for Scopes. FilterChains function very much like Unix firewall rules, e.g. iptables [Pur04]. They define a list of matcher rules for endpoints, in this case, operations on REST resources. The list is processed from top to bottom, and the first matching rule is applied (the remainder of the list is not processed). If none of the specified rules applies, a default policy is applied. Note that FilterChains can be held in whitelist or blacklist mode, that is the default policy can be `accept`, or `reject`. In general, blacklists are considered good practice, for newly added or changed endpoints then are by default protected. Since we are here only dealing with an authorization concern, which would be most likely combined with an additional restrictive mechanism for direct user access, the subsequent code illustrations reflect a whitelist (otherwise public endpoints would be no longer reachable at all).

In contrast to `@PreAuthorize` annotations, FilterChains are configured outside of the REST Controller classes, that is, in a separate file, as a Spring configuration bean. Listing 7.2 illustrates the securing of a REST operation for a given OAuth2 scope, this corresponds to a single rule, securing a single operation.

```
@Bean
SecurityFilterChain [...] {
  http [...] .authorizeHttpRequests((authorize) ->
    ↪              authorize.requestMatchers(
          HttpMethod.GET,
          "/bookstore/locations/{location}/stock/{isbn}")
          .hasAuthority("SCOPE_stock.read")
          .permitAll())
  [...]
  // Whitelist default rule, for all remaining
  // unprotected REST operations.
       .authorizeHttpRequests((authorize) -> authorize.
          ↪ anyRequest().permitAll())
  }
```

Listing 7.2: Spring Configuration to Enable Scope Verification via FilterChain

The parts in red in Listing 7.2 illustrate the matching of a resource (`/bookstore/locations/{location}/stock/{isbn}`) operation (`Get`) on a given required scope (`stock.read`).
The default blacklist policy is shown at the end, for completeness. Note that usually, the file would contain not only a single but a series of operation-specific scope requirements.

7.1.1 Variation Point Illustration

Next to the previously discussed definition of access restrictions (ownership and scopes), securing an existing REST service for OAuth2 also requires some initial boilerplate changes and configuration changes:

- For one, the application's build configuration (in the case of the sample application a maven `pom.xml` file) must be modified to declare dependencies to the OAuth2 technology used (Spring Security), so the aforementioned security annotations and FilterChain configurations are correctly interpreted.

- Secondly, the link to the Authorization Server (AS) must be configured. In essence, this simply corresponds to choosing between an existing, proprietary AS, e.g. provided by Google, GitHub, Meta, etc., or a reference to a self-configured and self-hosted AS. Spring does provide an off-the-shelf AS, a configuration of which is illustrated in the sample implementation repository.[1]

Similar to the variation point illustration in Chapter 6, we can capture the main decisions to be made throughout the conversion process of an existing, unsecured REST service as a three-stage process, as illustrated in Figure 7.2.

In summary, the process comes down to:

1. Orange: Build system boilerplate changes and initial AS configuration.

2. Red: Definition of Scopes and their operation coverage. Specification of ownership verification.

3. Blue: Service build and deployment.

The goal is now to build a CSL concern to streamline this process.

[1]Different ASs imply different token formats, which affect protocol security parameters. Tokens can e.g. be opaque, that is they require an extra introspection step on every RS API call. However, this is not the main interest of this concern, because Spring is by itself able to handle most common token formats.

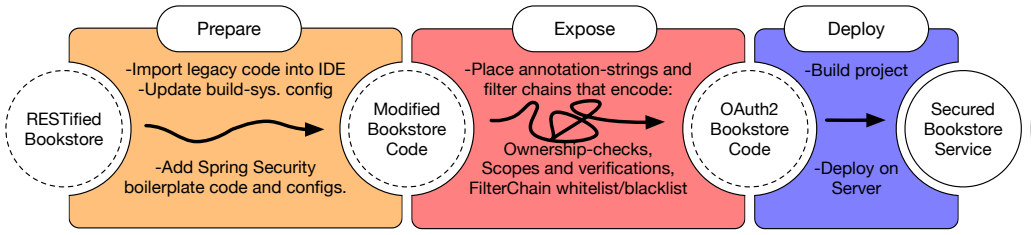

Figure 7.2: Manual Steps Required for Adding an OAuth2 Restricted Client Support to Existing REST Service Code

7.2 Designing the AUTHify Concern

In this section, I lay out how *FIDDLR*'s 3-staged plan of action is applied to the creation of the *AUTHify* concern. That is, I show the CSL meta-model, I define which mappings to existing model artifacts are allowed, and I define the model and mapping transformations needed to connect to the existing CORE and MDE pipeline.

Note that in contrast to the *RESTify* concern, the described steps for *AUTHify* have not been implemented, that is, *AUTHify* has not been integrated into the reference workbench TouchCORE due to time constraints. However, the bookstore sample code, which serves as sample concern output *is* fully implemented, functional, and validates the described pipeline stages.

7.2.1 AUTHify Variants

Similarly to *RESTify*, the first concern activity consists of creating a variation interface to capture the various AS options (step 1 in Figure 4.1). Implicitly, this means the concern creator also has to define the associated design models. As previously mentioned, depending on which configuration is selected, a corresponding RS configuration file (in yaml format) changes. This file notably contains information on how to connect to the AS.

7.2.2 CSL Definition

The main activity throughout concern creation is the provision of a new language definition, that is, the definition of the concern-contained CSL (step 2 in Figure 4.1). Once more this takes place using a meta-model. The interest of the language we propose, *AuthL*, is to provide a concise set of concepts needed to embody the characteristics of service securing with OAuth2. Note that the focus here lies on adding OAuth2 security concepts around an *ex-*

isting, unsecured service, not a full definition of a fully secured service from scratch. I will now briefly go over the language essentials, as depicted in the meta-model shown in Figure 7.3

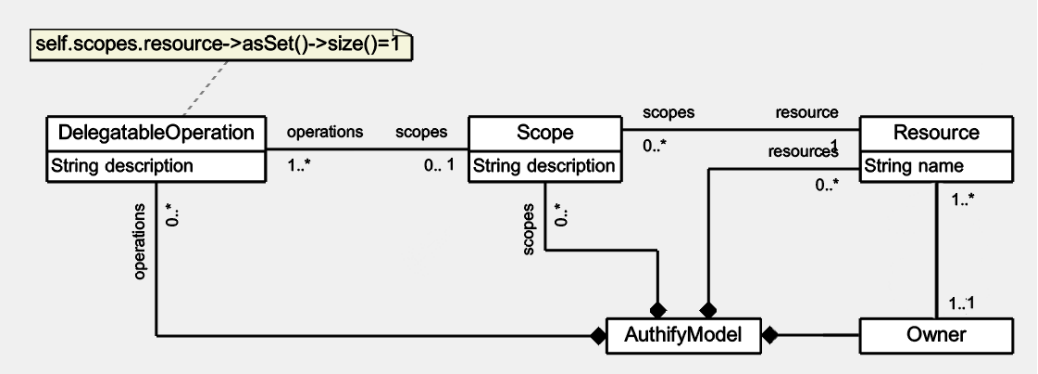

Figure 7.3: CSL Meta-Model for *AuthL*, of the *AUTHify* Concern

Since the meta-model's purpose is reduced to additional security concepts, it is relatively small. Still, it includes all additional concepts required for enabling OAuth2-based access delegation to an existing service. That is notably a notion for *Resources*, under the governance of an *Owner*, and of course *Delegatable Operations* on resources, which can be selectively covered by *Scopes*.
Note that the meta-model does not showcase scope overlap for a single operation, which is a design choice to enforce Resource Owner privacy. This is discussed in more detail in Chapter 8.

7.2.3 Mappings

Besides design models for all concern variants and the CSL language definition, a concern designer must also define how CSL concepts integrate with existing application contexts. This link is established by defining which concepts of the source and target language can be associated by *Language Element Mappings* (LEMs) [AMK22].
Compared to the previously presented *RESTify* concern, we now have to deal with two conceptual novelties. I will briefly point out each of them in more detail, and argue why the novelties are still in line with the *FIDDLR* framework.

1. The first LEM type to inspect in more detail are mappings from scopes to REST resource operations. As mentioned at the beginning of this chapter, we assume the service to secure has been previously restified, using the *RESTify* concern, and hence the unsecured interface to integrate with is a REST interface, partially expressed in the *ResTL* language. This means that our LEMs are now no longer from a CSL to a GPL, but from a CSL to another CSL. Namely, we are dealing with mappings from the *AUTHify* language's concepts to REST operations. For the mapping definitions, this is not an issue, as LEMs allow a linkage of any two target languages. But this consideration will come into play when it comes to model transformations, which I discuss in more detail in Chapter 8.

2. The second LEM type is for binding the notion of ownership to concepts of the application domain. In essence, this means we create a link between the CSL provided *Owner* concept and some entity in the application's domain model - typically expressed as GPL class diagram. The novelty here is that, along with the mappings to the *ResTL* language, we now deal with a concern that showcases polyglot mappings, that is coexisting mappings to models expressed in different target modelling languages: *ResTL* and class diagrams.

An additional consideration on LEMs in this concern mapping context is that they do not only define what can be mapped. The *absence* of LEMS can also serve as mapping information. The latter is just as important, as preventing concept mappings that cannot be meaningfully combined is a strong guiding factor for subsequent concern reuse. The LEMs discussed above for instance do not allow associating the CSL's resource identifier with a REST operation - simply because there is no meaningful semantic interpretation in establishing such a mapping.

7.2.4 Transformers

With the variants' design models, the CSL and the LEMs defined, the missing piece is model transformations. Those are needed to translate mapped models created in the previously defined *AuthL* language to standard GPL models, alongside composition specifications, so the existing standard CORE pipeline can take care of weaving (models of the same kind) and subsequent code generation. This is perfectly doable, following the *FIDDLR* plan of action, for the framework foresees two kinds of transformations:

CSL to GPL language transformations: Translation of *AuthL* concepts to GPL models (step 3 in Figure 4.1). For every secured REST operation, the transformer creates a stub class with the same method signature as the operation to secure and a `@PreAuthorize` annotation. The annotation payload is a generated SpEL expression, resulting from the mapping between the identifier resource, owner, and the intermediate path through the application's domain model (a visual illustration is provided in the next section on concern reuse). Note that this step requires deep mapping inspection. That is to say not only information from the target *ResTL* model is needed, but also information from the application context the *ResTL* model itself internally maps to (this has been described in the previous Chapter 6. Namely, the information needed is the exact GPL method signature to create, and the method parameter serving as the ownership identifier in the generated SpEL expression. The matching of the SpEL identifier on the operation parameter is illustrated by the red characters in Listing 7.1, where `location` is an identifier extracted from deep mapping inspection. The same holds for the `getStock` method signature, which is not contained in the *ResTL* model, but the GPL model the underlying *RESTify* concern itself maps to. Figure 7.4 illustrates the generated models and mappings, and how they are subsequently consumed by a standard GPL weaver.
Furthermore, the transformer must translate the scope coverage definition into a GPL model. This simply means the generation of a complete security configuration Spring bean, with the entire FilterChain. Every FilterChain entry reflects one secured operation, restricted to the required scopes.

Mapping transformations: The interest of the produced mappings is to consume the original CSL-CSL mapping information and produce GPL-GPL mapping information consumable by the CORE model weaver (step 4 in Figure 4.1). An important insight is that the sets of input and output mappings are not necessarily of the same size. This is for two reasons:

1. Some of the mappings are e.g. already consumed throughout the *ResTL* CSL to GPL conversion, described in the previous section. This is notably the case for the generated SpEL expression, which requires deep mapping inspection.[2]

[2]Note that CORE-provided GPL meta-models do not foresee semantics for annotation payloads. This is why the SpEL expressiveness, and notably the ownership notifier cannot be captured with a dedicated concept for ulterior mapping and weaving. In CORE's meta-model, all annotation payloads are static strings, which implicitly means they must

2. Some of the mapped CSL model concepts do not require mapping, once converted to GPL. This is for instance the case with GPL models for FilterChains. Models of FilterChain configurations do not require model weaving, as their existence in the generated class code is detected by Spring's annotation scan and interpreted.

In summary, the only output mappings required for *AUTHify* are operation mappings. Namely, these are method mappings, from all the previously created method signatures of stub classes to the corresponding REST annotated methods (so the weaver can combine the annotations), and mappings from the SpEL fragments representing method parameters to the corresponding method parameters in the target REST method.

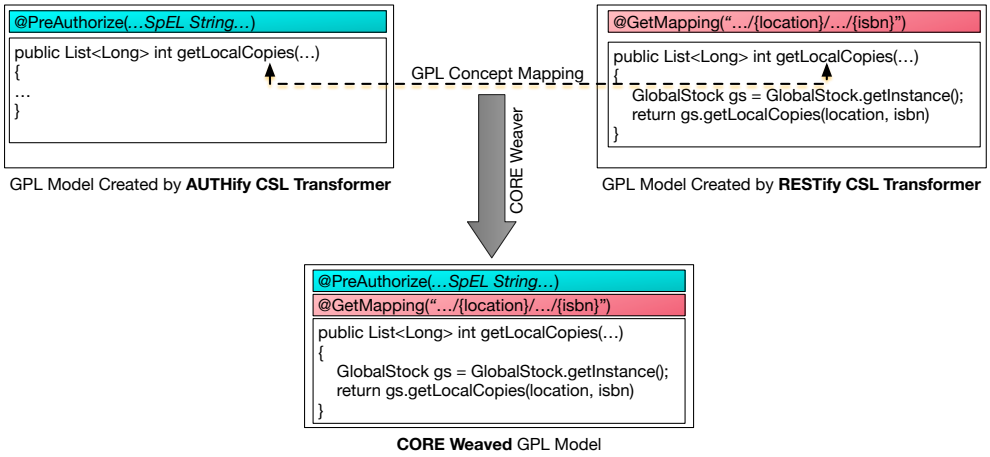

Figure 7.4: Illustration of CSL Transformer Generated GPL Models and Mappings

Figure 7.4 illustrates the code and mapping information generated for `@PreAuthorize` annotations, and how it is consumed by the weaver, to generate the GPL model of a fully secured REST operation. These operations occur entirely on the GPL models. The code listed in Figure 7.4 is semantically equivalent.

As a final note, the above transformer description also demonstrates why *AUTHify*'s mapping on a CSL language is not relevant for the concern-internal transformations. Model weaving is not provided by the concern,

be fully generated by the CSL to GPL transformer. Thus, this distribution of tasks is not *FIDDLR* derived, but rather a result of CORE.

but reused from the standard tools offered by CORE. As long as source and target models are of the same GPL model type and there is a valid mapping in between, the existing CORE toolchain can be reused as is. Since the target CSL stems from a previously defined concern, which in turn contains the required model transformation toward GPL models, this can be safely excluded from the *AUTHify* concern. Specifically, *RESTify* already contains a description of how *ResTL* concepts (which the *AUTHify* concern maps to) are translated to GPL concepts. This translation description can be readily reused. Additional illustrations for this process are depicted in Chapter 8.

Figure 7.5 illustrates how the described concern design activities integrate with the reusable tooling provided by the *FIDDLR* framework.

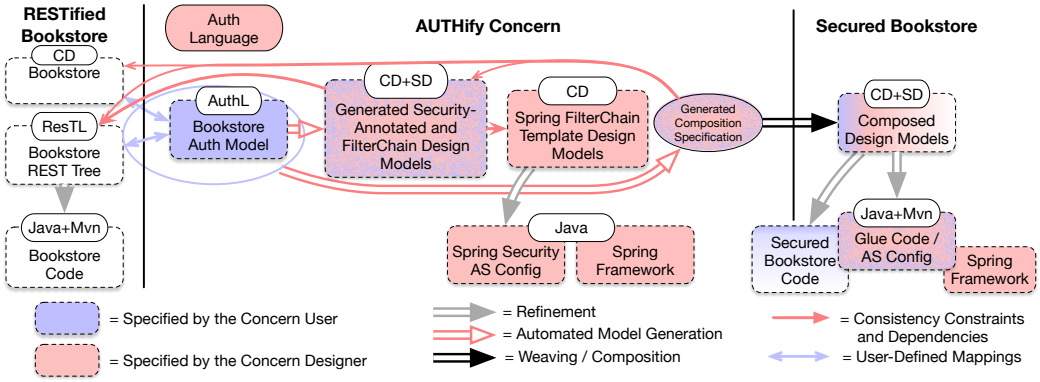

Figure 7.5: *FIDDLR* Applied to the *AUTHify* Concern

Namely, the four concern designer activities are:

1. Provision of Variants and variant-specific Authorization Server configurations: *Spring Security AS Config*
2. Definition of a custom concern-specific language, including LEMs: *Auth Language*
3. Definition of CSL to GPL transformers: *automated model generation*
4. Definition of LEM to composition specification transformers: *GPL model weaver instructions generation*

Table 7.1 maps these concern designer activities directly on the original framework steps, as illustrated in Figure 4.1.

FIDDLR Step	*RESTify* Concern Designer Element
Step 1: Realization Models	Design Classes, Maven Code
Step 2: CSL Definition	Provide *AUTHify* Meta Model
Step 3: Model Transformers	*AUTHify* to Annotations + Filterchain
Step 4: Mapping Transformers	*AUTHify* Mappings to Weaver Instr.

Table 7.1: Mapping *AUTHify* Design Actions on *FIDDLR*

7.3 Applying the *AUTHify* Concern

Reuse of the *AUTHify* concern follows CORE's standard three dimensions of reuse [KMA+16]: VCU, or Variation, Customization, Usage. Similar to the section equivalent for *RESTify*, I now present how these steps are perceived from a concern user perspective.

7.3.1 Variant Selection

The first choice to make when securing an existing REST API with the *AUTHify* concern, is which Authorization Server (AS) variant to use. As previously mentioned, the choice pertains to either going with an existing, already deployed proprietary AS, or starting with an off-the-shelf spring-provided implementation for self-hosting. While AS code generation is not within the scope of this concern, the choice matters, for the token format (and therefore the security standards) and communication strategy to apply by the Resource Server (RS) is reflected in an RS configuration file. Figure 7.6 illustrates the selection choices, as experienced throughout initial Variation Interface usage.

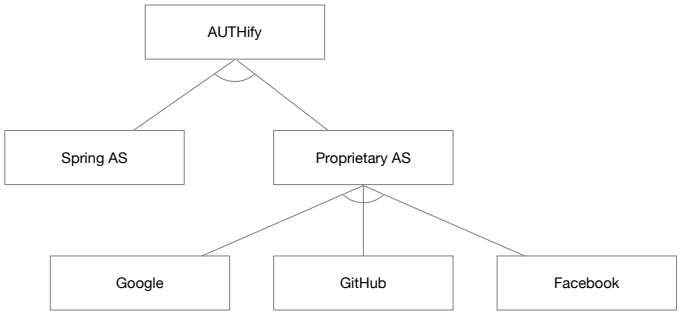

Figure 7.6: AUTHify Variation Interface Definition

7.3.2 CSL Modelling and Mapping

Independent of which AS has been chosen as a token provider, the next step consists of defining which operations on the REST tree are access protected, with a notion of ownership, which scopes exist, and how those scopes cover the protected resource operations. Here the concern provided CSL comes into play, to actively guide the user to express the required security constraints. Visually, the choices are represented by a triple-split view. Note that generic split-views have been developed as part of this PhD Thesis, a triple split view can be decomposed to a series of two individual split-views [SLL+21]. Figure 7.7 shows a *conceptual* editor for scope definition and operation mapping of the BookStore example. An important note here is, that in the classic manual process, Scopes are only defined in the FilterChain as string values, with no safety mechanism to prevent misspelling across repeated filterchain entries. This is notably not the case here, since a single scope concept is associated with multiple REST operations. What in the classic, code centrist approach relies on the correct, unchecked spelling of annotation payload strings, turns through the MDE approach into a semantic-first, visual activity.by

The upper part of the illustration defines scopes (left) and how they map to protected resource operations (right). Furthermore, the left side provides a concept for the resource owner (represented as stickman), who is eligible to delegate access to the defined scopes. Also, the left side depicts a concept representing resource owner governance (*Resource* box). This resource concept connects to a fraction of the resource tree on the right. The mapping defines for which fraction of the resource tree ownership is defined. For illustration purposes, I additionally highlighted the corresponding resource sub-tree with a pink contour and labelled it *location owned*. The lower split-view part represents the BookStore's domain model, which is either extracted or originates a base application model. Both, the resource owner concept, and the resource concept connect to the domain model, which defines a traversal. By passing through the domain model, we define how resource owner identity correlates to ownership, and therefore a resource on the *ResTL* model. This traversal can be translated into a SpEL expression, to verify if the resource owner matches on a targeted resource. Finally, the "+" element allows the concern user to create additional mappings, and the "i" serves as a popup for contextual information on the operation purpose.

The final step is unchanged, the service must be compiled and deployed. Just like in the previous *RESTify* example, this is very straightforward, as

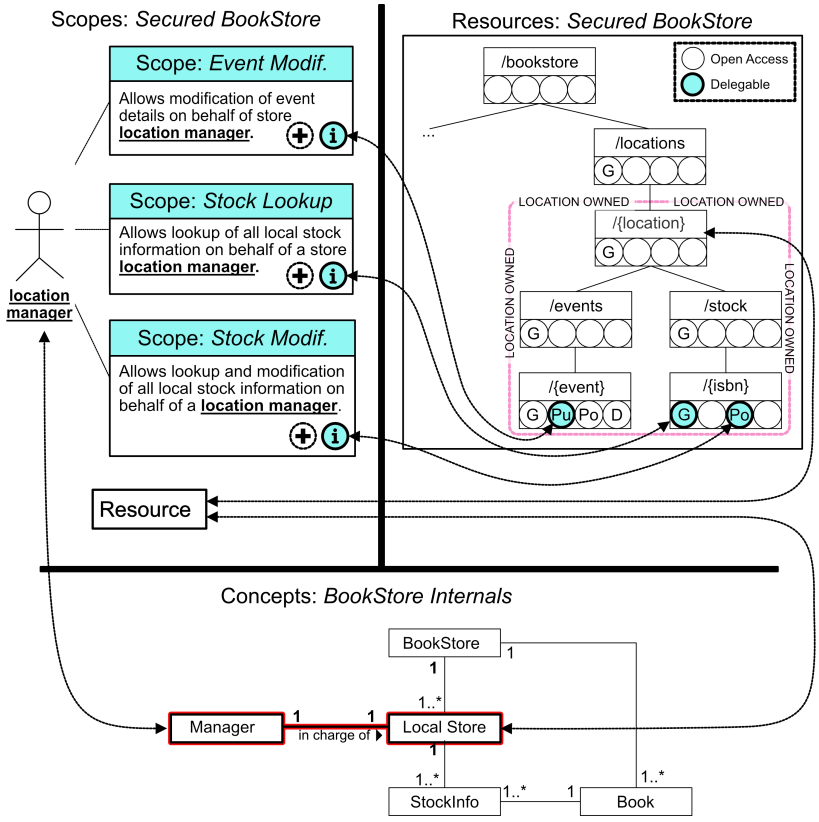

Figure 7.7: Conceptual Split View for Mapping Instance of the AUTHify CSL to the Application Context

long as the build system has been sufficiently configured to produce a self-contained executable. This step completely reuses the efforts already made for the *RESTify* concern, for it relies on the identical build system mechanism.

In summary, the described process greatly simplifies the key decision-making steps of the securing process, for the technical intricacies are concealed behind intuitive model representations. For the individual steps, *AUTHify*'s concern's contribution can be visualized as a simplification, as illustrated in Figure 7.8.

Notably the first two phases, which in the manual approach are tedious and error-prone, have been significantly streamlined, thanks to the concern provided variation interface, CSL, mappings, and model transformations.

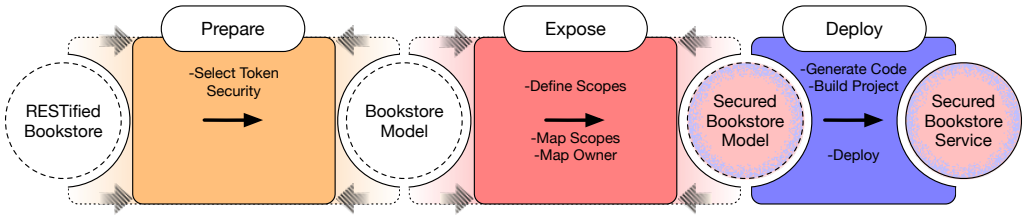

Figure 7.8: Application of the AUTHify Concern. Preliminary Tasks are Reduced to a Minimum. Decision-Making is Explicit

Unlike in the previous chapter, I do not provide a purely SLOC-based breakdown of the refactoring process. The SLOC metric would be out of place in this case, for e.g. the `@PreAuthorize` payloads are just a single line but can become arbitrarily complex in length.

An important note at this point is that, for lack of an actual concern implementation, the described advantages are prospective. Unlike *RESTify*, which has been incorporated into TouchCORE as a reference CSL-concern implementation, I was not able to test the prospective model transformations or generate code. Yet, I was able to validate the concern outcome, since I achieved a manual AUTHification of *AUTHify* the BookStore, which served as the target point for all code generations, and implicitly also all model transformations described throughout this chapter. A statement that can be made with confidence is that manually securing the bookstore was a complex, error-prone, time-intensive, and challenging task. Any model-driven help I could have had along the way would have been greatly appreciated. This first-hand experience is a strong argument in favour of the *AUTHify* concern.

7.4 Modelling Considerations

Service authorization mechanisms, and notably the OAuth2 protocol are inherently complex and must deal with many corner cases. While defining the *AUTHify* sample concern, and *AuthL* language, I oriented on the most common use cases. However, throughout the concern design two decision points were particularly debatable. In this final section I reiterate the choices made, and provide reasons that speak in favour or against the presented metamodel.

7.4.1 Scope Overlaps and Hierarchies

An interesting observation about the meta-model is the zero-to-one association between operations and scopes. Technically the OAuth2 protocol does not prohibit a definition of overlapping or nested scopes. Why would hence the CSL not allow for more complex scenarios like scope inheritance or multi-inheritance? Notably, why prevent such models, when established existing APIs with OAuth2 support, e.g. the GitHub API [Git23] showcase such scope hierarchies?
From a technical perspective, it would be little effort to modify the presented meta-model to support such scope inheritance scenarios, e.g. a *Read & Write* scope, extending an existing *Read* scope. Likewise generating the corresponding FilterChain code would be straightforward, for in case of an overlap, any scope would be accepted and Springs security syntax provides a mechanism, specifically for this purpose, as shown in Listing 7.3.

```
@Bean
SecurityFilterChain [...] {
  http [...] .authorizeHttpRequests((authorize) ->
    ↪              authorize.requestMatchers(
          HttpMethod.GET,
          "/bookstore/locations/{location}/stock/{isbn}")
          .hasAnyAuthority("SCOPE_stock.read",
              "SCOPE_stock.read-and-write"))
          .permitAll())
  [...]
  }
```

Listing 7.3: Illustration of Spring Security hasAnyAuthority Validation

Regarding the privilege outcome, there is no semantic loss in expressiveness between combinations of finer-grained scopes, compared to allowing equivalent inherited scopes. This is illustrated in Figure 7.9.[3]From a purely technical perspective, the lower model allows the expression of all combinations covered by the upper.

There is, however, a non-technical consideration to rule out scope overlap, and that is protecting the *Resource Owner Privacy*. While the protocol does foresee an option for the Resource Owner to reduce the set of requested scopes

[3]Granting B, which contains A is equivalent to granting A and C, which complement another to the same outcome.

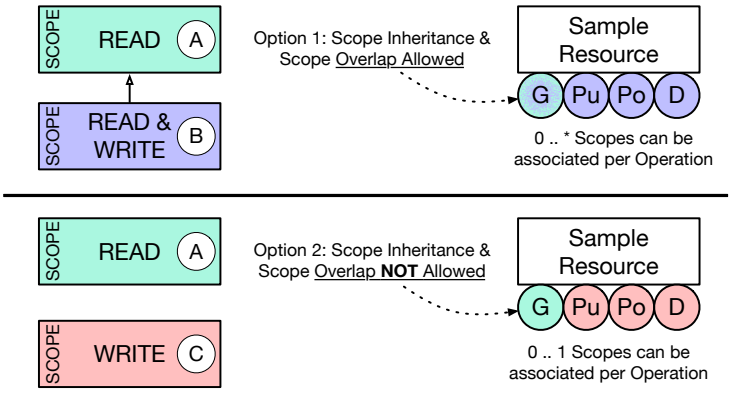

Figure 7.9: Illustration of Semantic Equivalence: A Client with Scopes A+C has the Same Access Rights as a Client with only B

at the moment of authorization[4], the protocol does not allow redefinition of scopes by the Resource owner [JR17]. That is to say, the only option from a RO perspective, is to turn down a subset of the requested scopes. In the worst case, this comes down to an all-or-nothing choice, e.g. if a Client requests full permission, by a corresponding scope at the lowest inheritance level, the RO can only entirely turn down the entire request, most likely resulting in a dysfunctional service. If on the other hand, the scopes are fine-grained enough to selectively disable inappropriate scope requests, a privacy-aware RO can maintain their interest.

7.4.2 Shared Resource Owners

The vast majority of OAuth2-ready Resource Servers consider the Resource Owner a single physical person. This holds, e.g., for a social media account, or web account, where a set of resources is under the strict governance of a single user. In that case, verifying ownership is rather straightforward, the RS simply compares the user's account name or identifier against the root of the owned resource sub-tree. In Section 7.2 I have shown examples of SpEL expressions to support this comparison.
However, there are cases where ownership is not restricted to a single user. The GitHub API is once more a prominent example, for GitHub supports

[4]This protocol option was not implemented by the sample dialogue of Obscrify, shown in Figure 5.6. A privacy-aware RO should be able to turn down the request for *Create, edit and follow playlists*, requested by a service that only serves for analytic activity.

organization accounts, with several members who each may or may not hold a notion of ownership in the spirit of OAuth2 [Git23]. We considered this scenario for the BookStore example, and the *AUTHify* concern can handle this use case. Similar to the GitHub organization reference, we considered that a BookStore location could be associated with multiple employees, each holding the right to delegate access authorizations on behalf of the organization. Regarding the previously presented editor, this does not require any changes, as illustrated in Figure 7.10.

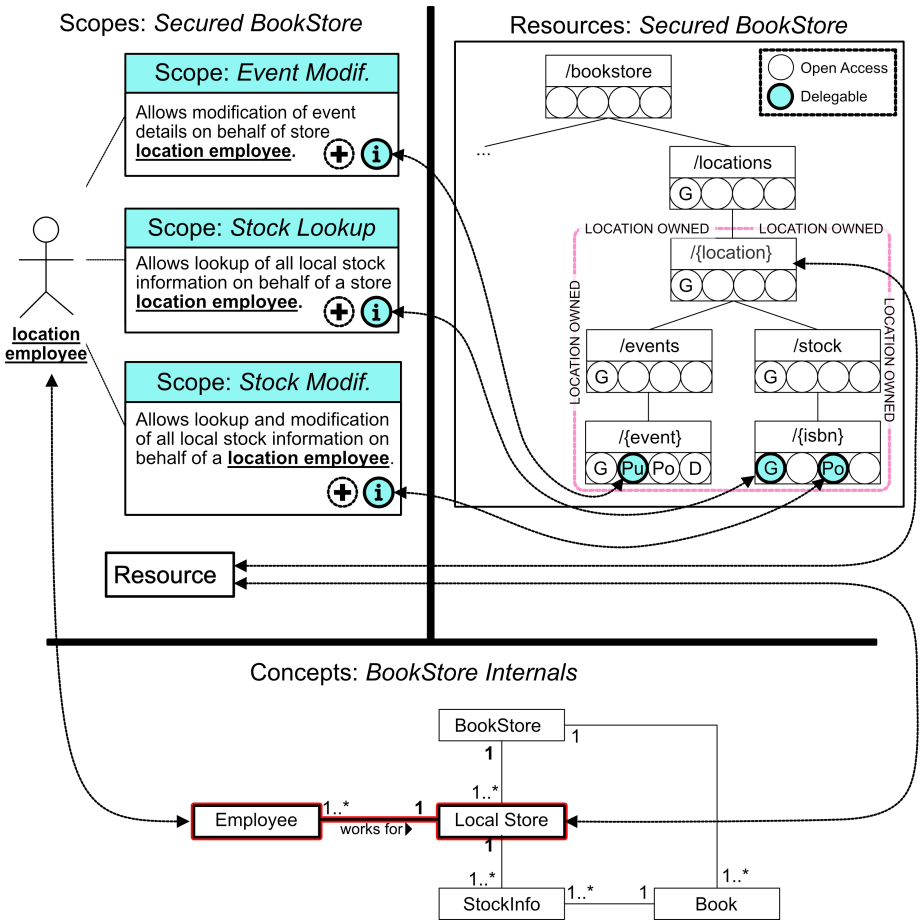

Figure 7.10: Conceptual Split View for Mapping Instance of the *AUTHify* CSL to the Application Context

Compared to the original sample editor capture, the changes are only marginal. Yet they reflect an important semantic difference in the application domain meta-model. Instead of iterating from one entity to the next, using a strict `1:1` mapping of the manager concept to the store location (serving as identifier resource in Figure 7.7), the association in Figure 7.10 showcases a `1..*` multiplicity, which in turn reflects the potentially shared ownership across multiple location employees.
The solution is once more to translate the domain model path into a SpEL expression, wrapped into a Spring Security `@PreAuthorize(...)` annotation: `"T(GlobalStockImpl)`
`.getInstance().getAllEmployeesForStore(#city)`
`.contains(authentication.name)"`
This string simply uses SpEL to traverse the domain model exactly as indicated by the red path in the split-view editor. The one-to-many association is reflected as a `contains` statement, meaning that any employee is accepted, as long as they are associated with the target store location.

Finally, it is noteworthy that ownership is a paradigm, that is, any criteria can be defined as a notion of ownership. This means the SpEL expressions (which otherwise need to be manually typed) can easily become arbitrarily long, and error-prone as the application domain grows. I argue that in this case, a visual representation of the actual domain model traversal path is a welcomed support for most software developers.

The takeaway message from this final section is, that when it comes to meta-modelling there is not one ground truth. Defining a guiding language that guides concern reuse is implicitly bound by user goals, ultimately resulting in a trade-off. The privacy vs. convenience example, when it comes to scope definitions is a strong example.

8

Takeaways from CSL Proof-of-Concept Implementations

The exploration of the two previously presented proof of concept CSL concerns, *RESTify* and *AUTHify*, kindled passionate discussions regarding the implicit nature and good design principles of CSLs. In this Chapter, I summarize the most outstanding observations and considerations.

8.1 Variants and Weaving

Throughout the previous chapters, we've seen several examples where the concern user's VI selection implies the inclusion of different design models and transformers. In this section, I illustrate this in more detail on the example of *RESTify*.

In Chapter 5, I have briefly mentioned the existence of various Java REST technologies, and that the same RESTful functionality can be reached in different means. Later, in Chapter 6, I have shown how one technology choice, namely choosing Spring Boot requires the integration of specific concern-provided realization models and transformations, notably to ensure the framework is integrated into the build system configuration, the launcher

boilerplate code is reflected in the design models, but also that the correct REST annotations are generated.
An essential observation is that the exact set of build a configuration, design models and transformer behaviour is tied to the technology selected in the VI. That is to say, when the concern user selects a different concern variant, the build configuration slightly changes, other design models are reused, and the transformers may need to generate annotations in a different syntax.
Especially regarding the build configuration the difference is subtle, for we do not simply need to generate an entirely different build configuration - it is rather that huge parts of the build configuration stay the same and only selected parts differ, depending on the REST technology chosen.
This incites reflection on how to best express reuse, based on the overlaps and similarities in the concern transformers for individual concern variants.

In the case of *RESTify*, we therefore tested model and transformer reuse across different concern variants. That is to say, we attempted to encapsulate commonalities between different concern variants and implemented the reuse pipeline to weave common, reusable parts with concern-specific model fragments. Figure 8.1 illustrates how this AOM technique manifested in the *RESTify* reference implementation. The different concern variants (colours) influence which design models (boxes) and which model transformations (arrows) are used for concern reuse. Black arrows indicate the reuse of generic design models, that are shared across all concern variants.

The left half represents the integration of design models and model generations for specific REST annotation syntaxes. The outcome of this first step is weaved RAM models (Reusable Aspect Models), which are a file format of the CORE reference implementation used. Note that these models, while adhering to a specific annotation syntax do not encode REST technology specifics.
The right half of the figure then indicates how partial build configuration models (of specific REST technologies) or woven with common build configuration ingredients. The outcome is technology-specific launcher configurations and technology-specific build system configurations.

Although we did not test the *AUTHify* concern to the implementation level, we validated the same effect on our reference authorization sample application. In the case of *AUTHify*, the generated Resource Server code showcases a configuration file that differs in specific lines, depending on which Authorization Server configuration is selected. The remainder of the file is

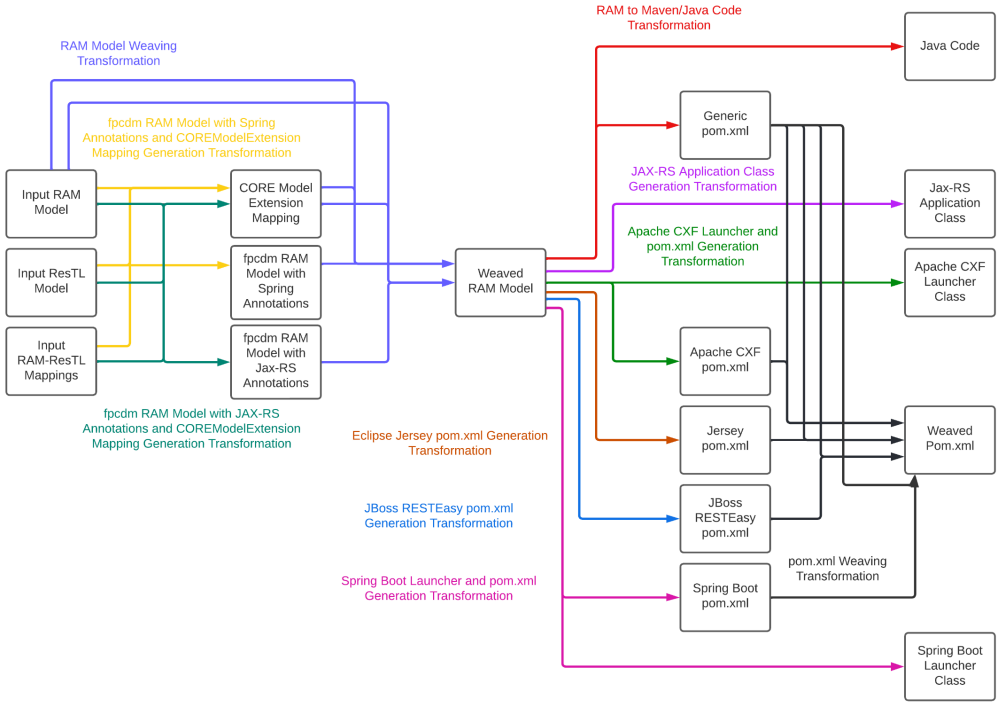

Figure 8.1: Illustration of Various Design Models and Transformers for Different Concern Variants. This Figure was Created by Bowen Lee as part of his Master thesis under the supervision of Maximilian Schiedermeier

identical. In short, this means, that in *AUTHify*, too, there is evidence for variant-specific configuration changes to justify configuration file weaving at the moment of concern reuse.

8.2 Abstraction Level

Common to both samples, *RESTify* and *AUTHify*, is the provision of a novel language, which allows the creation of tailored models to map on the existing application context. However, a substantial difference is that models created in *ResTL* map on GPL models, namely class diagrams, whereas *AuthL* models map on *ResTL*, which is itself a CSL. This implicitly means, *AUTHify* cannot be used without access to *ResTL* language definition, or the custom language tooling. This is illustrated in Figure 8.2.

Almost certainly this also means the *AUTHify* concern is bound to reuse the *RESTify* concern [1a] and is not viable in isolation. Internally, weaving the information from *AuthL* concepts into an application context works the same way, the CSL model is first translated to a GPL version [2a] (alongside mappings [2b]). The subsequent weaving however takes only place after GPL weaving with the *RESTify*'s intermediate GPL models, before the final code generation takes place.

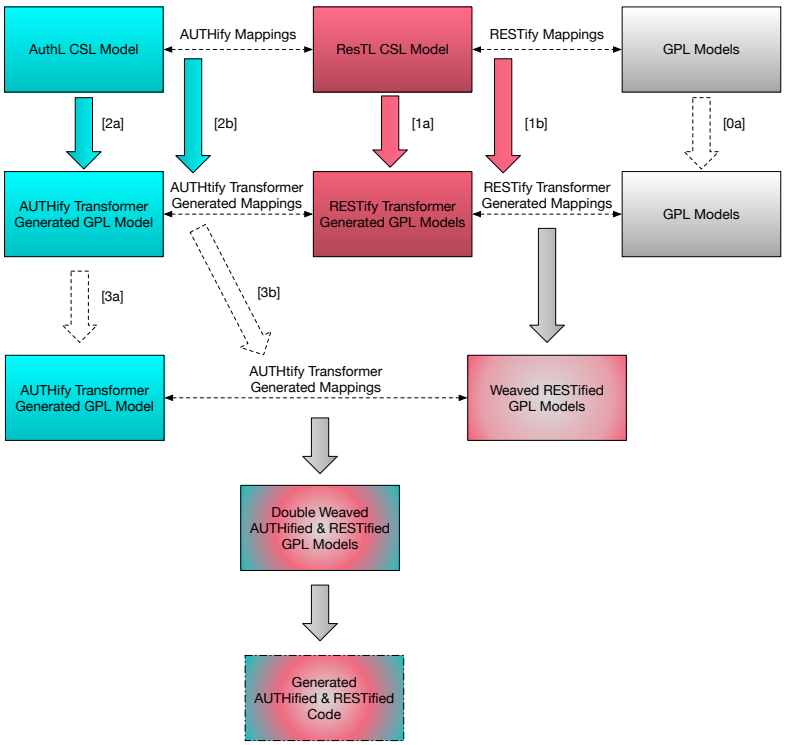

Figure 8.2: Cascading CSL Model Transformations and Weaving

8.3 Paradigm Mappings and Transformations

A peculiar observation in both sample concerns is the complexity of included transformers. Especially for annotation payload, we can observe that mapping transformations are more than simple one-by-one processing of input mappings to output mappings, to enable subsequent weaving. It is rather that some mappings are directly consumed by the CSL to GPL model conversion itself, and others completely disappear when creating the mapping

information required for weaving. While this is uncommon in the light of classic CORE concerns, this showcases a stunning coherence to the original CSL motivation: The philosophy that CSL concerns unfolds its full potential in the context of paradigm mappings.
This observation can be explained when we reconsider that semantic, or abstract equivalence is not the same as technical or implementation equivalence. A CLS concern targets the first, that is, it sets on visual paradigm mappings, ultimately creating a visual illusion of semantic equivalence, where there is no technical equivalence. REST interfaces are APIs so there is semantic equivalence to existing API structures, but that does not mean they are technically equivalent to classic Java method APIs.
Taking a step further is even the very nature of competing paradigms to be inherently incompatible at a technical level, which means they can only be readily mapped at a conceptual level. To give an example: a for-loop in an object-oriented language can be considered semantically equivalent to a recursive function call. Subsequently, we can create a visual mapping, to graphically represent a paradigm mapping. But that does not mean the underlying concepts are identical, or even readily compatible for direct weaving. CSL concerns require complex transformers because they must bridge semantic equivalence to actual technical compatibility.

8.4 CSL Design Principles

Several philosophies can be considered when it comes to defining a CSL metamodel. As discussed in the previous section, depending on which priorities are set, CSLs, and their corresponding editor change in size and apparel. CSL design can be guided either by striving for minimal, lean languages tailored to the CSL task, or the design can be guided by future language reuse and CSL extensions in mind. It is not clear if one philosophy is necessarily better.

- In the spirit of the original motivation, the focus can be set on SoC and model reuse as paradigm interests. Following that philosophy, CSL models should be viable in isolation, without the information expressed by mappings on one or more target models. Reuse scenarios here consider e.g. the possibility of reusing a part of a *ResTL* model, or a given arrangement of *AuthL* scopes, from one context to the other. To some extent this means information must be replicated, e.g. in a *ResTL* model. Replication was avoided in the case of *ResTL*, i.e. *ResTL* is only viable after additional mappings are provided, to infer e.g.

type information on parameters, and even body payloads. Inclusion of these concepts in the CSL itself would have been possible, and would, in turn, have rendered the CSL more self-contained. But likely a more wholesome CSL introduces also a need for additional checks during mapping, to ensure model and mapping consistency.

- Another consideration is to strive for lean languages, that by design sidestep replication of information. That is, language design can be guided by the philosophy that a CSL should not reify information already contained in the target model's concepts, to avoid inconsistencies upon mapping. Ultimately this allows the concern user to focus exclusively on the novel paradigm information that needs to be mapped, leading to simpler user interfaces, which is generally considered a fundamental design goal [Joh24]. The result is a smaller language, that focuses only on the concepts needed to achieve the given engineering task of interest for the present concern. Following this philosophy, *RESTify* should not provide type information for REST parameters, and *AuthL* should not provide information on operations covered by scopes - simply because this information can be inferred from mappings established to an existing target model.

It is unclear which of the two philosophies should be preferred. For our two sample CSL, *RESTify* tends to orient more toward the second ratio (it showcases only minimal replication and mappings to an application context provide additional semantics and are not just mappings of identical concepts). *AUTHify* is more influenced by the first philosophy, the replication of target concepts, namely operations and their semantics is part of the CSL meta-model, while the mappings to a target application context are merely matching instructions and provide no additional semantics. Both CSL are part of valid PoC concern samples. Finding a meaningful trade-off is dependent on the notion of a unit of reuse. If reuse of complete or partial CSL models is as primary goal, then CSL should be self-expressive to an extent, where mapping on a target model provides only contextual weaving information, but carries no inherent semantic. If the CSL and not the created models are considered the actual unit of reuse (because alongside the specification for legal mappings, they reify an expert's ability to create and explore models for a given paradigm), then the CSL should be held minimal. Means to further investigate this rather philosophical question are discussed in Chapter 14.3.

Part III

Empiric Assessment of CSL Effects

This third thesis part targets an empirical validation of initially presumed CSL toolchain benefits. I describe layout and results of a controlled experiment with humans to measure the effects of a sample CSL concern in action.

9

Background

Empirical research originates in a medical context, i.e., the testing of drug treatments on patients to gain insights on the drug's effects [CCS01, CS11]. Over the last decades, established methodologies have been adopted by Computer Science researchers, leading to a series of best practices and frameworks for the correct application of empirical research methods in our field. Nonetheless, the shortness of empirical evidence is an acknowledged ongoing issue in the MDE community [HWRK11, MCM14, HWR14, CMH19]. However, due to the origins, the medical terminology is still widely applied, which often renders empirical computer science research hard to consume for the broader community. For this reason, I now provide the fundamentals and terminology background needed for the empirical methodology applied in this last thesis part.

9.1 Crossover Layouts

Empirical studies usually follow a categorized design, that is in simpler terms to say a dedicated arrangement of participants and tasks. If chosen with care, a fitting design facilitates the entire experimental conduct, observation and later analysis and interpretation of observations. Throughout this last thesis part, I will be working with a design variant of the so-called *crossover layout* [CCS01].

In a crossover layout study, subsequent phases of treatments and observations (measures) are conducted on the same subject (study participant). These phases are also called *periods*. In the medical context, this corresponds, e.g., to testing two different drugs on a patient. Obtaining more samples from a population of a given size can be beneficial for the subsequent analysis - simply because there is more data collected. Insufficient data may not allow to conclusively confirm an effect, even if in reality it exists (this problem is also known as type-II error) [CCS01]. Another motivation in the medical context is the implicit pairing of data. Since multiple observations come from the same patient, i.e., the same biological organism, the paired samples are easily compared. Finally, since the order of treatments may matter, studies following a cross-over design usually consider all possible *sequences* of periods.

9.2 Carryover, Blocking Variables and Factorial Design

An important consideration for any repeated measures experiment is carryover. Carryover means unintentionally measuring the effects of the treatment of the previous period. If, e.g., first an experimental drug and afterwards a placebo were used as treatments, the observation for the placebo period can be influenced by lasting effects of the previous drug treatment. In medicine, this is handled with a *washout* delay, which simply means waiting some time for the organism to reliably recover from any effects of a previous treatment. Unfortunately *washout* does not exist in Computer Science, because humans do not reliably forget knowledge [VAJ16].

A common strategy to overcome this issue is the introduction of a *blocking variable* [CS11]. In the context of crossover layouts, this is referred to as a contextual *object* that changes over time. For instance, Ceccato et al.[CDPF+13] investigated two software obfuscation strategies (treatments). Each strategy was applied to a different software application (object), so that the knowledge obtained in the first period did not have a persisting influence on the second period. This combination of treatment and object is called a *factorial crossover design*. The *object* (or application to work with) appears in blocks in the layout table. If it takes two values, it is furthermore called a *two level* blocking variable. Table 9.1 illustrates the resulting experimental design. Note that the introduction of a blocking variable also increases the number of sequences to 4.

Experimental Design	**Period 1**	**Period 2**
Sequence I	Treatment A, Object 1	Treatment B, Object 2
Sequence II	Treatment B, Object 1	Treatment A, Object 2
Sequence III	Treatment A, Object 2	Treatment B, Object 1
Sequence IV	Treatment B, Object 2	Treatment A, Object 1

Table 9.1: Two-Treatment Factorial Crossover Design with Object as Two-Level Blocking Variable

9.3 Statistical Analysis

Best practices for crossover layouts also cover guidelines for the analysis of observations regarding the use of legitimate statistical tests [VAJ16]. Typically, an experiment seeks to determine relationships between variables (e.g., does taking the tested drug improve subject health). To do so, a *null hypothesis* is formulated, which in the vast majority of cases assumes a non-correlation of treatment and effect. If the outcome of the statistical test rejects the hypothesis, then there is a significant correlation. Note that only rejection is a conclusive result - failure to reject a null hypothesis does in return not mean that the hypothesis is true (it might, e.g., be that there is simply insufficient data for a statistically significant conclusion) [WHH03].

A further consideration is the eligibility of tests for the given data. Data from experiment observations can showcase several attributes, e.g., it can be paired or independent, the samples can be normally distributed or in the worst case, there is no knowledge about the underlying distribution. It is crucial to select tests that are compatible with the properties of collected data [VAJ16]. Some tests do, for instance, require paired data to operate on. In that case, they cannot be applied to experiment layouts that do not produce paired data.

Finally, in the case of factorial crossover layouts, it is recommended to consider and investigate the effect of all potential explanatory factors, not just that of the treatment. Linear models are a common choice for this purpose, for they allow correlation testing of multiple factors simultaneously, to compare how individual design factors contribute proportionally to the observed effects.

10

Experimental Design

In this section, I will now place the previously presented background on empirical research in the context of the RESTify Experiment, which is a study I performed to empirically assess the efficiency of the *RESTify* CSL.

As with all empirical experiments, the first step consists of defining the experimental design to use for our experiment. For this purpose, I start with translating the previously presented general crossover terminology to concrete study parameters. Namely, I delineate how the crossover layout is applied to assess the effects of *ResTL* on the conversion of legacy applications to REST.[1] Table 10.1 illustrates the experiment layout. At the centre of the experiment lies the comparison of two orthogonal conversion techniques. One of them is based on the *ResTL* CSL in combination with its tooling, the other is the manual software conversion approach, where subjects (participants) directly modify provided source code using an IDE. In our study, the alternative drug treatment of a medical context translates to the conversion technique applied.

As briefly mentioned in the background section, in contrast to the original medicine context, we cannot apply a washout period, where we wait for the effects of a previous treatment to disappear, before we run a second task and

[1] The exact study tasks are explained shorty when I delve into the individual experiment sequences.

repeat the measurement. Hence our participants must not work in the same context for their two respective tasks. Related work, e.g. a study on software obfuscation techniques by *Ceccato et al.* [CDPF+13] uses a factorial design with two different applications as objects (two-level blocking variable), which is a fair choice, for it eliminates the risk for carryover. We therefore apply the same design for our purposes.
This combination of a conversion technique paired with an application to convert now defines a software development *task*. One period of the study consists of the participant performing the task and us obtaining data from the process. Each participant partakes in two periods, which corresponds to two subsequent conversion tasks. Finally, as we want to combine each conversion technique with either application and since we want each series of tasks to be different in both factors (technique and application), we end up with a total of four sequences. For fairness, we want to attribute the same number of participants to each sequence, which is why we divide our population into four equally sized groups (labelled red, green, blue and yellow).

Experimental Design	**Task 1**	**Task 2**
Group I (Red)	CSL Conversion, BookStore	Manual Conversion, Xox
Group II (Green)	Manual Conversion, BookStore	CSL Conversion, Xox
Group III (Blue)	CSL Conversion, Xox	Manual Conversion, BookStore
Group IV (Yellow)	Manual Conversion, Xox	CSL Conversion, BookStore

Table 10.1: Two-Treatment Factorial Crossover Design as Applied in Our Study

10.1 Treatments (Conversion Techniques)

In this experiment, treatment is defined as the conversion technique applied by study participants. Our experimental design showcases two techniques: CSL-based conversion and manual conversion.

- **CSL-based Conversion**: The CSL-oriented approach does not require any manual coding. The project's legacy sources are interpreted by the TouchCORE modelling tool, which then visualizes the program structure as class diagrams. This corresponds to the concern reuse process detailed in Chapter 6. The developer then uses the REST-specific *ResTL* CSL editor to graphically design the desired REST resource tree. Afterwards, they graphically map those resource operations to individual methods of the legacy code. No additional action is required

from the developer, for the tool applies model transformation and composition techniques to combine the provided information and generate source code and configuration files.

- **Manual Conversion**: Manual conversion involves modifying legacy sources by hand. The developers open the existing project in their Integrated Developer Environment (IDE) and then import the REST technology of their choice. For the study, participants were encouraged, but not mandated, to use the same IDE as used in the task illustration material, namely the IntelliJ IDEA. After importing, REST-technology-specific annotations must be added to the code to expose operations. For our experiment, we only considered Spring Boot, which is a reasonable choice due to its widespread use. According to the *2018 JVM Ecosystem Report*, a survey based on 10,200 questionnaires, Spring Boot is the most popular Java web framework [MB18]. Correct integration of Spring also requires writing some boilerplate code and adjustment of project configuration files.

10.2 Objects (Sample Applications)

In our study, the objects correspond to two vanilla Java sample applications:

- **Object 1: The BookStore** We reused the source code for the previously presented Bookstore application [SKK21] that indexes books by ISBN and keeps reader feedback and stock information for stores.
- **Object 2: Xox** The second is an implementation of the simple board game Tic Tac Toe (referred to as "Xox", paraphrasing the *X*es and *O*s on the board). Xox is publicly available as an open source application [Sch22c].

We argue that both applications are representative study object candidates, as modern e-commerce applications as well as turn-based online games are predominantly implemented with RESTful backends. The two applications furthermore showcase a reasonable level of code complexity and size.

10.3 Periods (Tasks)

Each period consists of a conversion-to-REST task, defined by a combination of technique and application, allowing for an observation. That is, participants are presented with one conversion strategy, and one sample application, and we assess their task-solving. For the participants to perform a task unbiased, it is crucial to accurately and fairly describe what is expected. Therefore each period begins with a short task familiarization phase, where subjects only consume task material. Only afterwards they perform the requested conversion, where we collect data for a later analysis. We now describe task material presentation and data collection.

10.3.1 Material

The participants received the instructions for performing a conversion task in textual form, i.e., as a structured and easy-to-navigate website, and additionally through a video tutorial. Each video illustrates the expected conversion steps utilizing a third, independent example application. The steps are also explained in text and images on the website. Sound task material is important to give all participants the minimum required knowledge to perform the tasks. E.g., without detailed instructions, using the Spring framework could be daunting for inexperienced participants. Likewise, the instructions also contained textual documentation as well as a class diagram explaining the structure and functionality offered by the Bookstore, respectively Xox. Finally, the instructions describe the desired target state, i.e., details on the expected REST API behaviour after task completion.

A pilot study has shown that the two conversion tasks can be completed within two hours. We therefore allowed discontinuation after a minimum effort of 1 hour per task, including familiarization with task material.

10.3.2 Observations

We gathered two kinds of data for each task: produced *source code* and *screen recordings*.

- *Source code*, including build system configurations, represents the outcome of the application conversion. Depending on the conversion technique used, the source code is either the product of manual modifications or code generation.

- *Screen recording* covers all on-screen activity throughout the entire task. This notably covers familiarization with the instructive material, including watching the provided videos.

We also solicited participants to provide feedback after the completion of both study tasks. We asked the participants to fill out a text form, allowing them to provide comments on perceived complications, task complexity, and personal preferences.

10.4 Sequences (Groups)

A fair experimental crossover layout requires equally sized groups. In our case, this means that the same number of participants should perform the same tasks in the same sequence. In the remainder of this article we use the colours *red*, *green*, *blue* and *yellow*, to refer to the four equally-sized groups in our experiment.

To ensure group comparability we must also avoid any bias regarding developer skills. Experiments with a large number of participants usually rely on randomization to balance groups [CS11]. Larger sample sizes are also desirable because they reduce the risk of failing to conclude what is true due to insufficient samples (i.e., type-II errors). We did not perform a preliminary sample size estimation concerning this risk, because in our study the population size was in either case limited by the available funding. Our experiment with 28 participants is in line with comparative controlled experiments [KLB15]. Nonetheless, we applied additional measures to reach fair group partitioning.

10.4.1 Partitioning into Groups

Crossover experiments often include a preliminary observation to prevent potential group biases before task executions [CCS01]. We requested the participants to fill in a textual self-assessment form regarding experiment-related skills as part of the recruitment. Note that self-reported assessments, although by nature subjective, are common practice. Specifically, applicants were asked to declare their proficiency regarding the *Spring* framework, the *Maven* build system, the MDE tool *TouchCORE*, *Command Line* usage, the *REST* paradigm, the *Singleton* pattern, and *Reflection* (i.e., the programming language concept).
All these skills were potentially relevant for the study, and also potential predictors for success. We provided a textual metric to bring some objectiveness

to the declared level of proficiency on a scale of one to five. Listing 10.1 illustrates the process by presenting the questions regarding the *Singleton* pattern:

```
How much do you know about the singleton pattern?
1. [ ] I don't know what it is.
2. [ ] I know what it is, but have never used it.
3. [ ] I have already used it in one of my projects.
4. [ ] I could verify a provided implementation.
5. [ ] I could implement it right away from scratch.
```

Listing 10.1: Excerpt of Participant Skill Self-Assessment

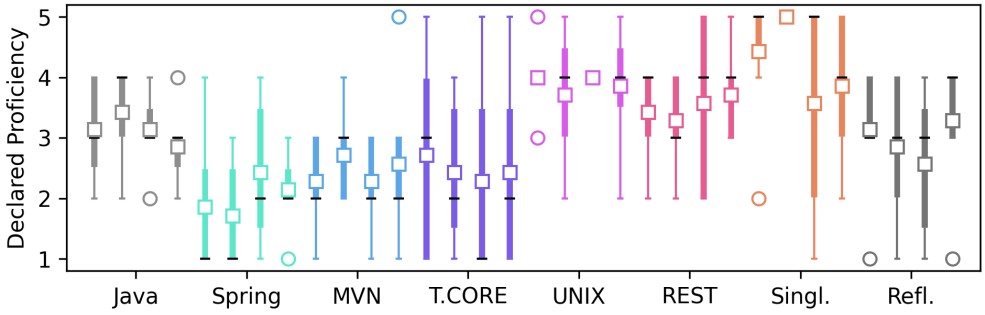

Figure 10.1: Skill Distributions Across Groups

I then implemented a heuristic to find a balanced group allocation. The algorithm minimized the distance in average skill proficiency between groups. Such algorithm is generally referred to as *MiniMax* heuristic, and while it does not produce the optimal participant allocation, it is less computationally intense than testing and ranking all possible allocations. With the given 28 participants brute force testing was not possible, for the amount of possibilities to test grows exponentially to the number of participants.
A severe challenge in this process was the repeated fluctuation of participants, i.e. participants retracting from the study after enrolment. The final allocation is the result of repeated recruitment iterations, and hence not the exact allocation initially proposed by the heuristic. Nonetheless, we obtained reasonably balanced groups as shown in Figure 10.1. Boxplots of the same colour represent the distribution of our four groups for a given skill. We observe that groups are reasonably comparable on all assessed skills. We further note that this balancing procedure is by nature not objective, for the initial data stems from a self-assessment.

10.5 Conduct

Recruitment and study conduct were carried out from June to August 2022. We launched an extensive recruitment campaign where we reached out to various targets: industrial engineers, engineering students and academic colleagues. We used various mailing lists, sent invitations directly, and recruited using a dedicated web page. Recruitment and experiment conduct was realized in full compliance with the experiment approval by McGill's Research Ethics Board (REB).[2]

We accepted all respondents, except when the preliminary self-assessment suggested insufficient programming skills for the successful conduct of the tasks. In the end, the 28 recruited participants come in approximately equal shares from academic, student and industrial backgrounds and showcased highly diverse skill sets. Participants were allowed to use their computer setup and participate remotely, with the option to access a lab-provided workstation as a fallback. Participation was rewarded with a 100 Canadian Dollar Amazon gift card.

[2]REB reference code: `REB-21-03-009`

11

Experimental Conduct and Analysis

Previously I have provided the experiment background, participant tasks and how groups were allocated. In this chapter, I present how I collected data, the statistical tests performed and which numeric results were obtained.

11.1 Preamble: Replication Package

An important preliminary note is that I have from the start worked toward transparency and easy replication of the entire experiment. Controlled experiments are subject to a plethora of parameters, and further research can only compare to this study if all parameters are known, and all analysis is easily replicable. To fulfil this goal I have created a detailed replication package [Sch23c]. The package contains, but is not limited to:

- Copy of recruitment material.
- Transcripts of all participant activity.
- Time measurements for all participant activity.
- All participant produced artifacts, notably their submissions, self-assessments and feedback.

- Indicators for outliers and submission patches.
- Task instructions, including original code, models and video material.
- Software used for testing of submissions.
- Interactive jupyter notebook to replicate all analysis and generated figures.

11.2 Data Collection

As presented in the previous Chapter, the experiment showcases two periods, which in this case simply means that each participant performed two subsequent tasks. I therefore obtained a total of 56 observations. Each observation consists of a screen recording and the produced source code or software models of the given task. When the outcome was a software model, I immediately generated the corresponding code to unify subsequent analysis. In the remainder, I also refer to screen recordings and software artifacts as raw submission data.

I carefully analyzed the raw data for two quantitative metrics: correctness of the produced software (by testing against the requested target REST interface), and *time* needed for execution of the task. Correctness is measured by the fraction of passed tests, a normalized metric. I refer to this metric as test *pass ratio*. Note that we are also interested in qualitative findings. In the following, I describe how I analyzed the produced source code and screen recordings.

11.2.1 Submission Testing

The task instructions contained a precise textual description of the expected target REST interface. This allows for automated testing of submissions. Note that RESTful services are stateful, therefore tests are interdependent. For example, a failed invocation of a "Delete" endpoint on a resource affects the result of a subsequent "Get" on the same resource. I avoided this issue by rebooting the tested service between every test call. Furthermore, tests that alter state require verification. It is not sufficient to check the HTTP return code of a "Put" request and assume that the service correctly modified the application's state. A subsequent "Get" is required to validate the behaviour. Finally, when verification using a subsequent "Get" fails, it does not necessarily mean that the initial state transfer failed. The failure can

also be due to an incorrect implementation of the "Get" request. I therefore invoked all state-modifying operations twice, once with verification and once without, to determine the cause of failure with certainty. Finally, in the one case where the read operation was determined to be incorrect, a manual code inspection had to be performed.

Figure 11.1 depicts the distributions of the *pass rates* per task. Groups are represented using colours, the conversion technique, the application and task order are shown as labels ("#1" = *first* period, "#2" = *second* period, whereas "#*" shows the distribution of the two groups combined). Turquoise and Orange elements indicate combined distributions of groups where only task order differs.

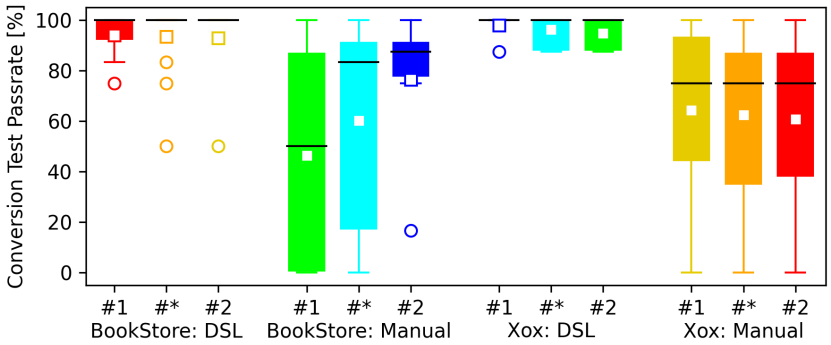

Figure 11.1: Distributions of Task Outcome *Pass Rates*

11.2.2 Screen Recording Analysis

I collected more than 72 hours of screen recordings, which I analyzed to determine: 1) the time the participant spent on task familiarization and task solving, and 2) unusual behaviour, task deviations, struggles and blocking errors.

Task *familiarization time* and task *solving time* were measured as follows: Any activity related to watching the instructive videos, as well as replication of the provided sample application (which was not part of the task) is counted toward familiarization time. However, as soon as a participant has completed a first pass of the instruction video, we consider all further watching of the video as targeted lookup and hence counted it toward task-solving time. The actual time spent on task preparation varies from participant to participant, for they often modified the playback speed and sometimes (although not requested) fully replicated the sample application before commencing the

actual task. Also, almost half the subjects deviated from linear order and interleaved familiarization and solving, i.e., pausing the video instructions prematurely to begin application conversion, switching back and forth between the two. In such cases, I manually measured the time for interleaved intervals to reconstruct the actual time spent on task familiarization and task solving. Luckily, task order was always respected.

Also, TouchCORE (the DSL tool) crashed in several recordings, which forced the participants to recreate the models already made, resulting in time losses. Note that we only use the task-solving times for the remaining analysis, as the purpose of this study is to measure DSL effects on software conversion, not on task familiarization.

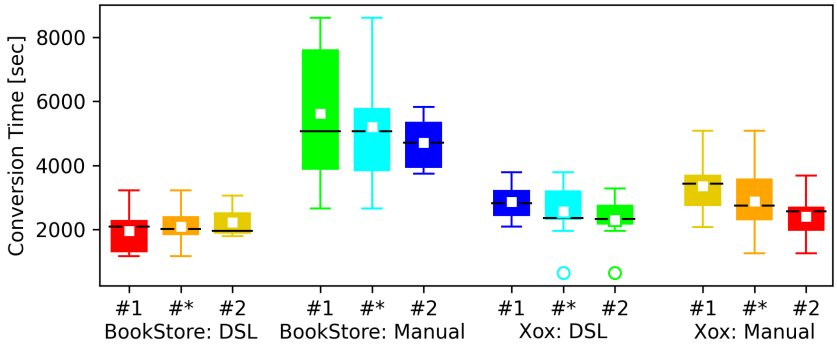

Figure 11.2: Distributions of Conversion *Times*

Figure 11.2 illustrates the measured conversion *time* distributions for each task.

11.3 General Linear Models

As mentioned in Chapter 9, it is recommended practice to begin analysis of data from a crossover experiment with a linear model [VAJ16]. I therefore started with dedicated models to investigate the effect of conversion technique and task order on the pass rate of the outcome and time needed for conversion. The general linear models (GLMs) used assume as null hypotheses that there are no effects of technique and order (also called *explanatory variables*) on pass rate and time (called *dependent variables*). We can then assess the likelihood of a non-correlation reported by the linear models, and ideally reject the assumed non-correlation for the conversion technique factor.

Note that our GLMs do not contain a dimension for the converted application (object). Instead, we have a separate GLM per application.[1] Consequently, I created two GLMs for *time* and two for *pass rate*, each based on a total of 27 independent sampling points.[2] Table 11.1 shows the outcomes for the resulting four GLMs, with *ordinance least squares* as regression fitting metric.

Dependent Var.	App	Prob. F-Stat	Explanatory Var.	Coef.	Std.Error	p-value
Pass Rate	BS	**0.018**	constant	53.90	9.94	74.41
			order	13.76	11.62	0.248
			treatment	32.67	11.62	**0.010**
	Xox	**0.020**	constant	64.21	9.52	**0.000**
			order	-3.43	11.14	0.761
			treatment	33.79	11.14	**0.006**
Task Time	BS	**1.97e-05**	constant	5332	449.9	**0.000**
			order	-292	526.3	0.584
			treatment	-3095	526	**0.000**
	Xox	0.059	(constant)	(3263)	(279.8)	(0.000)
			(order)	(-762)	(327.4)	(0.029)
			(treatment	(-288)	(327.4)	(0.388)

Table 11.1: General Linear Model Results for the four OLS Regressions

The first value to investigate is the reported F-Statistic probability, indicated as "P. F-Stat". It reports the goodness of our obtained model. The exact null hypothesis for this test is that a model with no independent variables fits the samples equally well or better. If the P. F-Stat value is below `0.05`, we can reject that assumption and in return retain the model present model sufficiently good for interpretation.

We first look at the two models (one per application) for pass rate. Both report a sufficiently good fit, we can therefore continue the interpretation of the individual factors. The *coefficient* for the "const" explanatory variable is not relevant for our interpretation as it only defines the overall offset of the determined linear regression. The coefficients for the other two variables *order* and *technique* however are interesting, as they define to which extent these factors contribute to the observed pass rate. We see that in both cases *technique* has a higher coefficient than *order*, which means that according

[1] This is necessary, because the samples from repeated measures are paired, and otherwise an additional factor would have been required to represent the participant, or at least group. In principle this would have been possible with a mixed linear model, however we did not have enough sampling points to reliably use this model type.

[2] 27, because one submission was a scam and not usable. Every GLM only deals with one application and, therefore only uses one sample per participant. Samples from the same participant are dependent, and we did not have enough samples to obtain a sound regression for two additional dimensions (participant and application) in a single model.

to the model, the conversion technique is a stronger predictor for pass rate than the task order. The coefficients alone are however not sufficient to conclude significance. We also have to investigate the *probability* that each factor correlates to pass rate (our explanatory variable). The linear model automatically performs a test for non-correlation and reports the probability for this null hypothesis as "P >|t|". Once more we can reject the assumption if the reported value lies below the "0.05" threshold, which holds for *technique*, but not *order* in both cases. In summary, we can conclude from the first two linear models that technique is a significant predictor for the measured pass rate.

Interpretation of the GLMs for *time* works the same. However, we notice that only the model for the BookStore reports an F-Statistic probability below `0.05`, while the model for Xox does not. This means the Xox model is not a good enough fit to continue interpretation. Only the model for the BookStore can be safely interpreted (since the reported value is only marginally above the threshold we will still interpret it, for completeness). The BookStore model is fairly similar to the two previously discussed models for *pass rate*. Once more, *technique* reports a higher coefficient compared to task order, and the technique factor is likewise the only explanatory variable for which the p-value rejects the null hypothesis of non-correlation.
While I am hesitant to consider the results for the remaining Xox *time* GLM, I still report the model interpretation for completeness. Interestingly this model lists the *period* (task order) variable with a higher coefficient than *technique*, and likewise only rejects non-correlation for *period* (order), but not for *technique*.

In summary, based on the model's self-reported "goodness", only three of the four linear models should be interpreted. These three models altogether suggest *technique* as a relevant predictor for the dependent variables *pass rate* and *time*. Likewise, these GLMs report a comparatively weaker coefficient for *order*, and also fail to reject the null hypothesis of non-correlation for the *order* explanatory variable.

11.4 Wilcoxon Rank Sum & Effect Size

Next, I perform an additional test to further investigate the significance of conversion technique as a significant predictor for pass rate and conversion time. Here I only focus on the conversion technique, and explicitly disregard a potential effect of order (note that *order* is not necessarily an irrelevant

factor, for the three sound GLMs only failed to reject a non-correlation, and the last GLM was rejected).

The methodology for this second test is as follows. Per application, we assume as a null hypothesis that matching sample distributions are identical, regardless of which conversation technique is applied. To illustrate this, we compare for instance the time distributions of all participants who manually converted the BookStore, to the time distributions of all participants who did the same using the DSL technique. If the null hypothesis is rejected, we know these distributions are different and have additional evidence that the conversion technique matters. We run a total of four comparisons, where we compare pass rate distributions and time distributions for both applications. Visually this corresponds to comparing pairs of orange and turquoise boxplot distributions (we ignore task order) in Figure 11.1 and Figure 11.2.

We applied the Wilcoxon Rank Sum test for these comparisons. This test was chosen because it is robust for small sample sizes, and does not make assumptions on the sample distribution [VAJ16] (we do not know the sample distribution, other than the preliminary boxplot visualization).

The p-values, in order of appearance of the above null hypothesis, are: **0.000016** (*time*, bookstore), 0.35 (*time*, xox), **0.00489** (*passrate* bookstore), **0.00389** (*passrate* xox). The corresponding interpretation is that for pass-rate measurements of BookStore and Xox, as well as for time measurements of the BookStore, the migration technique (treatment) causes a significant effect.

These p-values mean that the null hypothesis is rejected for all comparisons except conversion *time* of the Xox application. This means three distribution pairs should be considered different, whereas, for the last one, we cannot conclude distinctiveness. In return we can retain that conversion technique matters for the outcome pass rate (both applications) and likewise matters for the required time (for the BookStore).

These findings are sound, however, the tests only determined the distinctiveness of three distribution pairs but did not quantify their differences. Table 11.2 lists the numeric boxplot information for all compared sample distributions for pass rate and time.

A numeric comparison of the distributions serves as effect size estimation, i.e., a quantification of the measured offsets. In numeric comparison, we observe a relative improvement of averages whenever the DSL technique is applied instead of a manual code conversion. The numeric differences in distribution averages are as follows: For the BookStore an average pass rate

	BS: DSL (#∗) (Orange)	BS: Manual (#∗) (Turquoise)	Xox: DSL (#∗) (Turquoise)	Xox: Manual (#∗) (Orange)
Q4: Max (Upper whisker)	100.0% / 3060s	100.0% / 8615s	100.0% / 3602s	100.0% / 5084s
Q3: 75% Quartile	100.0% / 2439s	91.7% / 5835s	100.0% / 3043s	87.5% / 3619s
Q2: Median (Black Bar)	100.0% / 2021s	83.3% / 5069s	100.0% / 2357s	75.0% / 2754s
Average (White Square)	93.5% / 2091s	60.3% / 5197s	96.2% / 2564s	62.5% / 2881s
Q1: 25% Quartile	100.0% / 1769s	16.7% / 3789s	87.5% / 2096s	34.4% / 2250s
Q0: Min (Lower Whisker)	100.0% / 1167s	0.0% / 2651s	87.5% / 1955s	0.0% / 1251s

Table 11.2: Distribution Quartiles, Disregarding Order (#∗)

improvement from 60.3% to 93.5% (33.2% higher pass rate), at an average task speedup from 5197 to 2031 seconds (3166 seconds faster). For Xox an average pass rate improvement from 62.5% to 96.2% (34.3% higher pass rate), at an average task speedup from 2881 to 2500 seconds (381 seconds faster). In all cases, we also observe a lower variability of the distributions whenever the DSL technique is applied.

In conclusion, both analysis techniques (GLMs and Wilcoxon Rank Sum test) as well as the final effect size estimation draw a consistent picture. Both analyses suggest a measurable positive effect of the DSL technique for conversion times and test pass rates of the BookStore, as well as the test pass rates of the Xox application. Note that the null hypothesis of GLM and Wilcoxon Rank Sum has not been rejected for the Xox conversion time. This does not mean there is no effect, it only means we were not able to prove or disprove such an effect with the tests applied. The effect size estimation still shows a weak positive effect of the DSL technique for the resulting average Xox conversion time.

12

Interpretations and Discussion

In this chapter, I delve into an interpretation of the previously presented quantitative results. This notably means I will take into account contextual knowledge and additional qualitative observations. The latter stems from a thorough analysis of screen recordings and feedback forms. Note that this discussion only reports on the most outstanding findings. Afterwards, I discuss an observed discrepancy between the measured benefits of the DSL technique, compared to critical study feedback in Section 12.2. The chapter concludes with a compilation of several recommendations for DSL-based software development, suggesting how tools could mitigate existing drawbacks and improve practical DSL acceptance.

12.1 Understanding the Offsets

In the following, I discuss several factors that explain the performance offsets of the previous analysis. Specifically, I take a closer look at explanations for why the manual code conversion showcases a lower test pass rate than the comparative MDE-assisted counterpart. Secondly, I reason why we observe a slower manual code conversion for the BookStore, but not for Xox.

12.1.1 Lower Test Pass Rate for Manual Conversion

The previous statistical tests determined the conversion technique as a significant predictor of the pass rate. In other words, the source code produced by participants using the DSL approach is more likely to be correct.

The screen recordings showed that the participants had difficulties with Spring's annotation syntax. We often observed participants confusing `@PutMapping` with `@PostMapping`, or simply forgetting an intermediate resource in the URL path. This suggests, that the DSL approach is more intuitive, mainly because of the global resource visualization in the form of a tree and the automated generation of the annotations, which avoids the aforementioned errors. Most participants also reported the DSL approach as more intuitive in their feedback form. I provide a more detailed breakdown of the corresponding statistics in Section 12.2.

I also noticed issues related to the mapping of request parameters. Several subjects performed online searches for the correct annotation syntax and subsequently invoked study-unrelated parameter types. In more detail: The study only included resource parameters and body parameters, but no query parameters. Yet some participants, when searching for external resources, came across the query parameter syntax, and mistakenly added query parameters to their implementation. The effect is a deviation from the requested target API and consequently results in test failures.

Finally, we need to consider that generated code (automatically created by the DSL approach) is always syntactically correct. Manually created submissions could potentially not compile, which then implicitly means that all tests fail. However, all participants decided to work longer on the code until it compiles, than submitting code with compilation errors.[1]

12.1.2 Slower Manual Conversion for the BookStore

Similar to the test pass rate, our tests show a significant task speedup for one application (BookStore) when the DSL approach was used. Interestingly, the speedup is almost marginal for the second application (Xox). This observation is likewise coherent with the statistical test results. I first provide general reasons to explain a slower manual conversion, and then argue why the observed effect is lower for Xox.

[1]The only exception is the identified scammer, which has been excluded from the data used for statistical analysis.

In principle, time is a censored variable, because the participants were allowed to discontinue the study once the minimum time had passed. As previously mentioned, all participants simply continued their work until their code compiled. The vast majority also selectively tested REST access before submission. Code syntax errors occurred in all manual task-solving, and resolving the errors required additional time.

A second important factor that influenced completion time is the configuration of the *Maven* build system. The manual conversion requires textual modifications of the build system's `pom.xml` configuration file. This required adding a dependency statement to the Spring framework and updating the *Launcher* class information. Although the required changes were detailed in the instructions and available as copy-paste-ready configuration snippets, many participants reported issues with this step.

Likewise, the screen recording footage confirmed that participants frequently overlooked the *Launcher* class information, simply starting their submission using the IDE launcher symbol during development. Notably, this means their submitted solution was technically not compiled using maven at development time, for the launcher symbol by default does not invoke the build system configuration. While technically this causes all tests to fail, we decided to patch submissions that had this mistake, as this particular step is a pure boilerplate activity. Wherever we patched, I included a note in the manual submissions (part of the replication package). In contrast, the toolchain support of the DSL performs these configurations automatically, and the developer does not need to be concerned with them. This implicitly means that this build configuration challenge does not influence the measured pass rate statistics, but only the measured time.

Footage has also revealed a second issue related to the build system: changes to the configuration file are in some cases not automatically detected by the IDE. In those cases, since the build system also handles runtime dependencies, calls to the Spring framework, e.g., are highlighted as errors in the code editor. Although the code is correct, the IDE still bombards the developer with errors and warnings until a manual configuration refresh is done. For some participants, this slowed down the conversion progress.

Finally, I want to comment on why the difference in conversion time is larger for the BookStore than for Xox. I believe this can be well explained by a subtle (and unintended) difference in the nature of the two sample ap-

plications that drastically affected the complexity of the manual application conversion. The manual conversion requires applying Spring's *Dependency Injection* mechanism: if one class needs another class, Spring annotations can be added to highlight which dependency should be injected where. In the case of Xox, all functionality to expose over REST resides in a single class, which in turn meant there was no dependency injection to resolve. In the case of the BookStore, however, the functionality is distributed over three classes. These classes need to be wired using Spring's dependency injection. Although the concept is explained in the video instructions, 7 participants mentioned this issue in their form, and the screen recordings show that even more participants were struggling. Problems with dependency injection are critical, for incorrect configurations stall the application startup. Consequently, when dependency injection is not resolved, all tested endpoints fail. However, the screen recordings show that participants resolve this error in their manual conversion by investing additional time. As a result, only three manually converted submissions of the BookStore exhibited this issue. It is a fair assumption that all Spring applications above a certain size showcase dependency injection. Therefore, we think the offset measured for the BookStore reflects reality better than the one of Xox. The issue of Xox not requiring the use of dependency injection is further discussed in the threats to validity Chapter 13.

In several of the previous explanations, we connected the better results for the DSL technique to toolchain-related benefits, in particular to code generation, where boilerplate steps and modifications of configuration files are performed automatically by the toolchain. A fair question is to which extent the positive impact should be attributed to the DSL, rather than to the MDE toolchain. I argue that since DSLs are rarely viable in isolation, their benefits should not be disassociated from the general observed MDE effects, notably code generation. However, we would also like to point out that the tree-based DSL and the visual mapping of REST endpoints with methods in the source code led to fewer mistakes and lower conversion time compared to the manual insertion of annotations across the source code.

12.2 Perceptions and Bias

Overall, the participant feedback is consistent with observations from the screen recordings. However, there are interesting contrasts between the observed issues, reported issues, and measured offsets in the statistical analysis. That is to say, the issues mentioned by participants in their final comment forms do not perfectly align with qualitative and quantitative observations. There is an offset between subjectively perceived and objectively observed engineering challenges.

12.2.1 Issues with the DSL Conversion Technique

A frequent, but non-critical issue with the DLS toolchain technique was the TouchCORE tool itself. Participants repeatedly noted difficulties navigating the tool's menus and reported accidental deletion of already modelled solutions, which was confirmed by the recordings. This can be interpreted by insufficient intuitiveness of the tool's user interface.[2] For some participants this resulted in data loss. All participants eventually learned how to navigate the tool, and the time loss was never significant when parts of a solution were reestablished after a crash. To avoid biases we did not remove crash-inflicted slowdowns from the measurements. For completeness, I did however quantify the exact losses and included them in the replication package. Overall, we consider that the main challenges with the DSL technique were instability and usage of the tool's user interface. Both challenges could be resolved by improving the tool's reliability through additional testing and an interface revision guided by user studies. Nevertheless, despite these drawbacks, using the *ResTL* DSL led to overall faster task completion than the manual alternative.

12.2.2 Perceived Time Loss

As previously discussed, the major time losses observed and reported are problems with *Spring's Dependency Injection*, *Maven's configuration file editing* and using the *DSL tool's user interface*. However, interestingly a frequent mention was the task instructions themselves, notably the task illustration videos. More than two-thirds of the participants either skipped parts of the

[2] A strong argument for this hypothesis is the fact that TouchCORE was originally developed for touchscreen interfaces, and only later adapted for desktop use. The graphical interface, and especially user input handlers still showcase touch-oriented relics, e.g. a tap-and-hold gesture, which is very uncommon for desktop interfaces.

video, increased playback speed, or interleaved task solving with watching the video instructions. While the instructions were created with great care, video material naturally provides a one-size-fits-all level of detail. Very likely the instructions were perceived as overly lengthy by some participants, leading to the described effects. I will discuss in the next Chapter 13 how this affects the validity of our study.

12.2.3 Preference for Manual Conversion

We analyzed the final participant feedback regarding their preferences concerning both migration techniques. While the majority of participants deemed the DSL-driven approach easier (+25/1/-1) or more intuitive (+24/0/-3), the confidence towards a better performance of the DSL solution concerning our unit tests was balanced (+13/1/-13), and a majority of participants would not use the DSL technique for their own future projects (+6/6/-15). These statistics are illustrated in Figure 12.1.

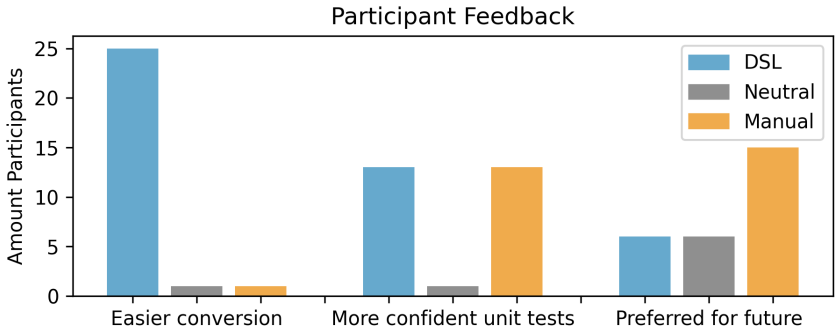

Figure 12.1: Participant Feedback on Individual Techniques

The trust in correctness significantly deviates from the measured correctness, and the stated preference seemingly contradicts the initial statement for intuitiveness. Yet, in the light of additional participant feedback, there is a plausible explanation for this trend. Several developers associated the manual approach to feeling more in control:

Green Unicorn: "*(I prefer) the manual solution, because it gives more control over the source code.*"
Green Turtle: "*I will most likely stick with IntelliJ as I feel more comfortable coding everything manually where I have more control.*"
Yellow Turtle: "*Because the code generation process is unknown to me, I'd be more confident in the manual methodology [...], where I had total control and knew the code that would run against the tests.*"

At the same time, the main reasons for mistrust in the DSL toolchain seemed to be the opacity of the model transformation and code generation. In some cases, the feedback would even first acknowledge the advantages of the DSL approach, but then state a preference for manual refactoring:

Yellow Fox: "*I'm always suspicious of auto-generated code*"
Yellow Zebra: "*(I'm more likely to apply) IntelliJ because it feels more natural [...].*"
Green Zebra: "*TouchCORE (DSL technique) is more intuitive as it is more visual, but the IntelliJ (manual) approach helps better to understand the underlying mechanism.*" and: "*(With the manual approach) I know clearer what is going on behind the scenes compared with the second (DSL) approach.*"
Blue Zebra: "*There's a lot of boilerplate code in RESTifying a legacy application, TouchCORE (the DLS tool) makes this easier and less error-prone.*" and: "*(I'm more likely to apply) manual, I've had problems with code generation tools in the past*"

In summary, the developers acknowledge the merits of the DSL approach, yet tend to prefer the manual approach. The main reasons are 1) the association of coding with "being in control" paired with overconfidence in their coding skills in comparison to their actual test results, and 2) a general mistrust in opaque transformations, especially in code generators.

12.3 Towards Practical DSL Acceptance

I believe the insights from the discussions are an important finger post towards further improvement of DSL-based approaches. The user feedback suggests acceptance is not exclusively a matter of benefits, but also of tooling apparel. Based on the full feedback received, and observations from the video, this bias could be mitigated with the following action plan:

12.3.1 Traceability and Transparency

Any SE toolchain should be as transparent as possible. Developers trust the compiler, partially due to the obvious link from code statements to execution instructions. Using a debugger, that link is highly transparent, and users tend to forget the existence of implicit transformations. In comparison, the model transformation and code generation taking place in *ResTL* are opaque, and thus hard to follow by the user. A more advanced implementation should provide clearer, immediate traceability of modelling choices to the generated code, to re-establish the developer's perception of control.

12.3.2 Integration over Disruption

Developers consider their preferred IDE a trusted environment and are naturally reluctant to abandon their comfort zone. Rather than proposing an orthogonal approach that effectively replaces the entire IDE, a DSL toolchain

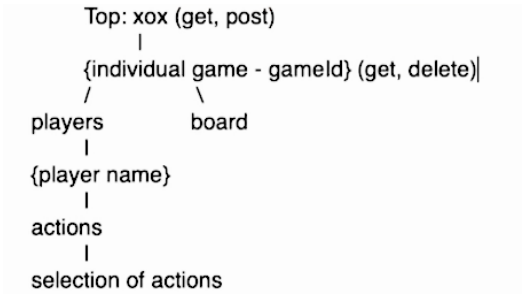

Figure 12.2: Imitating the *ResTL* DSL in a Text Editor

should integrate with the existing, trusted terrain. Multiple participants voiced they were more likely to embrace the benefits of the DSL-based approach if the toolchain were integrated as an IDE plugin, rather than a standalone software. This option combines well with the previous point, as a plugin could more easily highlight the impact of modelling on generated code. Interestingly, we collected anecdotal evidence about a participant's desire to have the *ResTL* language available within their IDE. The participant in question, who had used the DSL-based technique for the first task, created an ASCII imitation of the *ResTL* DSL while working on the second, manual conversion. Figure 12.2 shows a capture of their recorded activity.

12.3.3 Trust through UX

While the DSL and mapping process was predominantly perceived as highly intuitive, the user interface and stability of the toolchain were criticized. Especially the crashes and the gesture-based user interface were perceived as tedious. While this did not diminish the outcome of the DSL-based conversion process, I believe that the instability of the tool influenced trust in the generated outcome. In comparison to an industrial-grade IDE, a DSL toolchain can only gain acceptance if stability and usability are excellent.

In summary, we believe the mistrust is not due to the DSL and the associated modelling and model transformation mechanisms themselves, but due to the way they are presented. Developers consider IDEs as their trusted workmode, so the most straightforward way to improve on acceptance would be to develop an IDE plugin that also includes mechanisms to visualize the resulting transformations in the source code. Such a tool could offset the factors that we associate with the observed developer bias and would allow us to leverage the DSL's currently dormant potential.

13

Threats to Validity

This section enumerates factors that potentially weaken the validity of the presented empirical findings, as well as the measures taken for mitigation. I structure the discussion according to four types of validity, i.e., construct validity, internal validity, conclusion validity and external validity [WRH+12].

13.1 Construct Validity

Duration of a software task and test pass-ratio are both established and common constructs to measure the success and efficiency of a software development task [Kan03, DEPC21]. We are only aware of one participant, who did not try to follow the instructions in a meaningful way. We considered the corresponding data and outlier and excluded it from analysis and interpretation.

13.2 Internal Validity

Internal validity concerns the degree of confidence that there are no hidden variables or factors affecting our measured pass rate and completion time. This aspect has been treated with great care, using multiple GLMs.

13.2.1 Varying Developer Skills

The varying skill levels of developers introduce bias to our measurements. I ran additional Pearson tests that did not report a statistically significant correlation between any of the self-reported skill levels and the measured time or pass rate.

13.2.2 Parameter Information

We noticed that the DSL toolchain in one aspect behaved differently on the Windows operating system. When mapping resources to methods, the participants using Windows would not see identifiers of target signature parameters, i.e., they would see the method identifier, the number and types of parameters, but not the names of the parameters. We deem this issue to be negligible because the missing information was provided in the task documentation. As confirmed by consulting the video footage, the slowdown for Windows participants is negligible. More precisely, video footage showed that participants rapidly consulted the provided source code or documentation to overcome any ambiguity.

13.2.3 Task Deviations

The experiment was carefully designed to guide participants in their endeavours. As mentioned in Chapter 12, the majority of participants did not consume the video instructions as intended, increasing playback speed, skipping parts or even interleaving listening to the video instructions with solving the task. As a result, some participants initially started to convert the illustrative Zoo application that was used for explanation purposes in the video. Others experienced slowdowns because they had to deal with technical problems whose solutions were clearly explained in the instructions.

It is hard to quantify to which extent this phenomenon influenced our findings. Additional Pearson tests (which I provide in my replication package) did not report a significant correlation between task familiarization and task-solving time or pass rate. Although not statistically significant, we observed in a few cases that sloppy task familiarization suggests negative effects on both techniques.

13.2.4 Fair Task Description

The task instructions should not favour one of the conversion techniques. To ensure that, we could neither provide the REST interface description visually in tree form (as this would have favoured the DSL approach), nor as textual URLs (as this would have simplified the manual approach). We therefore used an intermediate textual representation that we deem unbiased.

13.2.5 Fair Task Context

We acknowledge an unintentional difference in task complexity because Xox did not exhibit the aforementioned dependency injection challenge. Luckily, this drawback does not weaken the relevance of our findings, because with increased complexity the performance of the manual conversion could have only been lower. For all other metrics, e.g. magnitude of interface size or code base complexity, the Bookstore and Xox are highly comparable.

13.3 Conclusion Validity

Conclusion validity concerns the statistical analysis of results. Small populations jeopardize significance, for outliers gain impact. We countered this threat with intense recruitment, having a population of 28 participants, respectively 7 participants per experiment group. Regarding our hypothesis, we have consistently employed tests and followed best practices to ensure that the assumptions of the used statistical analysis techniques are met.

13.4 External Validity

The largest threat to the external validity, that is, the generalization of our experiment results to other contexts, is the recruited population. Skill distributions of students and professions are known to differ, and the use of experiments only with students is often debated regarding external validity [FZB+18]. We mitigated this risk with efforts to recruit diverse developer profiles, i.e., we were sending out invitations to developers from various backgrounds. Some factors are beyond our control, e.g. industrial engineers might feel less attracted by the compensation offered. We considered that this may cause our participants to be younger than average software developers, and thus possibly less experienced. However, the collected self-assessment forms do not reflect such a bias.

Part IV

Conclusion, Discussion and Future Work

In this fourth and last thesis part I recapitulate the previous contributions and findings. I place the individual conclusions into context and sketch the bigger picture of CSLs as a promising novel MDE concept.

14

Conclusion, Discussion and Future Work

In this final discussion, I pertain to the cornerstones of my thesis, present a fair summary of limitations to my work, and indicate how future research could further extend the aforementioned boundaries.

14.1 Conclusion

This section summarizes the main contents and insights from the first three thesis parts, in order of appearance. That is, I begin with summarizing the key takeaways from the *FIDDLR* framework, followed by lessons learned from crafting the two sample concerns and finally, the insights gained from my controlled *RESTify* experiment.

In the first part I presented *FIDDLR*, a framework to streamline reuse and promote separation of concerns that integrates MDE, DSLs and CORE. *FIDDLR* augments the unit of reuse of CORE, the concern, with the possibility of including a modelling language that is specifically designed to express the concern's properties and integration most appropriately. In other words, the concern designer can now define a *Concern-Specific Modelling Language* to maximally focus the concern user on the relevant concepts of the concern and facilitate the concern's customization and usage. Just like DSLs, doing

this can significantly reduce accidental complexity as well as integration complexity, in particular for concerns that are not easily expressed with a GPL or that crosscut several abstraction levels or phases of software development.

In the second part, I delved into two sample CSL-driven Proof-of-Concept concerns. *RESTify* streamlines a state-of-the-art development activity: exposing application functionality as RESTful services. Thanks to *FIDDLR*, the concern designer implementing *RESTify* can reuse the existing MDE, DSL and AOM technology and tools at various levels in the development process. I showed how *RESTify* greatly facilitates the task of exposing application functionality as RESTful services for the concern user, compared to a manual refactoring activity. The second sample concern, *AUTHify*, builds on top of *RESTify* and allows securing unprotected REST services for access delegation of resource owners to third-party services, following the OAuth2 protocol. Similarly to *RESTify*, I showed how the concern-driven approach eliminates the majority of technical intricacies and guides the concern user toward explicit decision-making. Both concerns show convincing evidence that *FIDDLR* bears great potential and merits further investigation.

In the third part, I presented the details of a controlled experiment with 28 software developers that shed light on the impact of using a CSL for converting a legacy application to REST, and how using such a CSL was perceived by developers. We quantitatively compared two orthogonal conversion techniques: using a CSL and corresponding toolchain vs. manual code conversion. We observed a superior effectiveness of the CSL approach in terms of conversion speed and correctness of the outcome. We then discussed plausible explanations for the observed difference based on a thorough analysis of the recorded task activity and participant feedback. The developers encountered technical difficulties in the manual approach, in particular with the dependency injection mechanism of the Spring framework. Furthermore, some developers had trouble with boilerplate steps, i.e., updating the Maven configuration files and modifying the launcher class. This accounted for significant additional development time and reduced the correctness of the outcome sometimes significantly. These technical details are completely automated and hidden from the developers in the CSL approach. Furthermore, we found evidence that the developers considered the visual representation of the REST resources in the form of a tree as beneficial and intuitive. In contrast to the numeric findings, the analysis of the qualitative developer feedback revealed low acceptance of the CSL approach. We identified con-

fidence in code, and general mistrust of MDE transformations, especially code generators, as the main factors for preferring the manual approach. We used these insights to formulate recommendations for developers of CSLs and associated tools.

14.2 Discussion and Limitations

Although the presented contributions serve as convincing evidence for the general feasibility of CSL-enabled concerns, and their potential for the SE community, I also discovered several hindering factors that currently set limitations to what can be achieved with CSLs. This section explains the three most outstanding challenges in more detail.

14.2.1 On FIDDLR Limitations

FIDDLR streamlines reuse by reduction to four essential steps:

1. Choose a concern variant from the concern's VI.
2. Model the concern-specific properties of the application using the CSL.
3. Specify mappings that connect the concern-specific properties with the application-specific models.

As a result, the concern user is shielded from solution-specific design choices and technical intricacies. Furthermore, the complex transformation pipeline that FIDDLR is based on – CSL to GPL model and mapping generation, weaving, and code generation (see Fig. 2) – is also hidden from the concern user. This fundamental operating mode of FIDDLR is inherently linked to a general limitation: low traceability between what a concern user models and the generated code. The feedback from study participants and the general mistrust voiced regarding black box code generators fall in line with this drawback. Furthermore, low traceability hinders debugging seriously and can make it very difficult for the concern user to apply corrective actions at the CSL level in situations where the generated outcome does not produce the expected behaviour. While traceability and debugging are well-known issues with generative approaches as well as with compositional approaches, the problem is even more pronounced in FIDDLR because it uses generation *and* composition technologies. A further limitation of our implementation of FIDDLR is that it was built on top of the CORE tooling infrastructure and therefore inherits all its technical limitations, i.e., CSLs have to be EMF-based, and code generation targets Java and Maven only.

14.2.2 On CSL Limitations

A severe drawback of CORE and especially CSL concerns is the level of expert knowledge required to craft new concerns. While concerns are intended to streamline reuse and make their application as simple as possible, their crafting naturally requires considerable domain expertise. This notably holds for CSLs, as in this case the concerns are not limited to mostly static off-the-shelf models but comprise entire languages. The PoC samples and following discussions in the second part of this thesis revealed that language design is indeed a complex enterprise. Also, the definition of transformers, to translate from mapped CSL models to mapped GPL models requires proficiency with domain and implementation details, to ensure the transformer output is sound for subsequent model weaving and code generation.
From previous practical experience with REST interfaces, the design of *RESTify* came easier as the following exploration of a second *AUTHify* concern. I see the amount of practical expertise and domain sovereignty required as a major hindering factor for the creation of new CSL concerns. Notably, meta-modelling in that context is a multi-scholar discipline, as it requires expertise in MDE and the targeted domain. Another observation is that meaningful CSL candidates are not easily identified. Following our insights from Chapter 8, it seems the most eligible candidates for CSL-supported concerns are engineering activities that showcase an inherent paradigm mapping (such as mapping resource-oriented API structures on existing signatures, or mapping of access limiting scopes as masks on operations). This observation is in line with the initial assumption that CSLs unfold their full potential in the context of crosscutting concerns, as paradigm mappings naturally bridge multiple levels of abstraction.

14.2.3 On PoC Implementation Limitations

Whatever the costs, striving for an industrial-grade implementation, and running a large-scale study, is in my perception the only way to fairly assess the potential of CSLs. Unfortunately, it seems that reaching an industrial-grade implementation is also almost certainly an extremely resource-intensive and pricey avenue. Even with the considerable amount of time invested throughout this thesis, the most advanced PoC implementation, *RESTify*, made it barely beyond what was minimally required to run a fairly limited user study in a mostly controlled setup. Pushing the envelope would require several iterations of user studies, and massive investment into general software

stability. The latter is a concern that is easily ignored in academic settings. The shortcut to declaring implementation issues as trivial or not relevant to research should not be made. A sophisticated analysis of present stability issues could likely reveal a series of fundamental limitations.

14.2.4 On Study Limitations

The quality of empirical validations is tightly dependent on the underlying sampling quality. That is for one, a large enough sample population size (or enough participants), but also a population that somewhat represents reality. Empirical research references provide protocols to ensure that these criteria are met. Namely, sample size estimations can be run preemptively, to ensure the population is large enough to ensure conclusive probability margins for the anticipated results. Additionally, it is common practice to capture population features like demographics or skills. In reality, the available funding is often the limiting factor to population size and constitution. The same holds for the experiment described in Chapter 8.4. I hired as many participants as possible with the available funding, and I did my best to ensure a diverse population, e.g., by including industrial engineers outside of academia. In retrospect, one mistake was to offer the same compensation for all target audiences, simply because the 100 CAD Amazon gift card was more attractive to undergraduate students than to industrial engineers, which by itself could already have constituted a biasing factor. And yet, in the end, the population obtained was larger and more diverse than what I had hoped for. Nonetheless, there were drawbacks, e.g. we could not follow some recommendations, such as using a mixed linear model, for lack of sufficient sampling points. To some extent, I was able to compensate for these drawbacks, e.g. by ensuring balanced groups instead of random assignments and selecting statistical tests that were compatible with our data. Overall the study was diligently performed, within the given limitations.

14.3 Future Work

In this final section, I provide several ideas on how the presented research could be taken further. The proposals are sorted by order of appearance of the corresponding item in the thesis outline.

14.3.1 Exploring the Nature of CSL Reuse

Throughout the design of *RESTify* and *AUTHify*, we mostly applied simplicity as the main design criteria for the concern-provided CSLs. As discussed

in Chapter 8, this is a justifiable rationale for decision-making, because lean languages are less prone to overwhelm the user, are faster to learn, and hence often perceived as more intuitive for the creation of on-point models.
A fundamental question throughout CSL design was whether CSLs should or should not replicate concepts that already exist in the target application context. An argument against concept replication is that CSLs are designed to unfold their full potential in combination with mappings to existing model contexts, and replication of information is not in the concern user's interest and potentially causes consistency challenges. On the other hand, overly reduced CSLs can hinder reuse scenarios, for isolated models (without mappings to an application context) do not carry enough semantics for reuse in different contexts. We could for instance imagine a scenario where a partial *ResTL* model fits well in a reusable context (e.g., common board game concepts), and could be a candidate for reuse across applications.
Although these two philosophies stand in stark contrast, it is hard to justify one over the other, for the advantages and inconveniences of either approach depend on the trajectory of future CSL concerns. With a growing CSL concern library, a focus on CSL reuse, possibly even across related concerns could be a fostering factor, whereas even our empirical research has shown that concern users long for easy-to-use and intuitive interfaces.
An interesting thought experiment on *RESTify* is, why a novel REST CSL needed to be defined in the first place. Any existing REST modelling language presented in 5 would have served, as any REST language would have readily provided the concepts for hierarchically arranged resources to be mapped. Notably, the outcome of *RESTify* is, next to deployable code, an interface description in one of these languages, namely OAS (the build instructions carry an instruction to analyze the target code's Spring annotations and generate the OAS interface specification). If the product of a reused concern is a complete REST interface description, there is no justification for a CSL of identical expressiveness. In future work, I would like to further investigate the differences between CSLs compared to regular DSLs.
The philosophies on the ideal wholesomeness of CSLs turn the discussion into a chicken-and-egg problem: More concern reference implementations are needed, and this could be simpler by putting an eye mark on CSL reuse. However, concern implementation is primarily justified by simplicity to the concern user, which speaks in favour of leaner CSLs. Further research could begin with a survey of multiple representative CSL candidates, to better assess the legitimacy of either design criteria and potential compromises.

14.3.2 Layered CSL Concerns

The *AUTHify* concern serves by definition a very specific purpose. As explained in Chapter 5, *AUTHify*'s use case is authorizing third party services, to access user-owned resources on their behalf. The reference implementation presented in Chapter 7 does for instance only deal with access delegated to authorized clients, but ignores the access control task, to authorize the actual owner to access their resource. This is because usually access delegation is not viable in isolation, but integrates with other, more standardized API security concepts, such as user roles (RBAC), ownership hierarchies and even shared ownership. The presented concern correctly supports the key protocol scenario. But to gain more viability it should probably be considered in combination with other prominent security concerns, possibly bringing in additional CSLs. Further research is needed to explore the model weaving required to support such CSL concern combinations.

14.3.3 Extended *AUTHify* Concern

Another consideration regarding the *AUTHify* concern is that in its current form, it only allows the securing of RESTful services crafted with the *RESTify* concern. This limitation stems from the fact that the required model mappings are defined from the *AUTHify* to the *RESTify* language. While from an academic point of view, it is a reasonable choice to investigate the effects of stacked CSL concerns, there is no real-world argument for this limitation. A versatile *AUTHify* concern should be able to readily secure any REST interface, regardless of how the service was created. In principle, the required modifications are very feasible. Similar to how we extract GPL models from existing code, we can extract OAS specifications from existing REST services. Consequently, it is very thinkable to modify the *AUTHify* concern, so it maps from the *AuthL* to an OAS model, sidestepping any dependency on the *RESTify* concern. However, there remains the challenge of tailored models and mapping transformers. It is not clear how such a revised concern could achieve the subsequent weaving process from mapped and extracted OAS models to produce secured RESTful service GPL models. More research is required to implement such a (more versatile) *AUTHify* concern.

14.3.4 CSL-IDE Integration

In Chapter 12, I showed statistics that many participants acknowledged the intuitiveness and convenience of the *RESTify* Concern, but simultaneously voiced reluctance to apply the tool for their own or future projects. A main reason was a stated preference for working with a trusted IDE, and the MDE approach being perceived as too disruptive. A seamless integration of *RESTify* with existing IDEs could not only leverage user acceptance but also unfold new potential. E.g. IntelliJ already offers a built-in class diagram editor, which always reflects the current code state. A redesigned version of *RESTify* could embed directly into the IDE as a freely available plugin, and integrate with the existing diagram editors. This would also allow enhanced mapping modes, where the *ResTL* model is alternatively mapped to class diagrams or directly on existing code signatures. Afterwards, the plugin could extract GPL models from the mapped code and apply the same transformation and weaving pipeline as the current *RESTify* concern implementation. Notably a freely available, open-access plugin could provide large-scale telemetry data, to gain further insight on common use cases and preferred usage scenarios for the concern.
Figure 14.1 shows a mockup of how the *ResTL* CSL could be used as assistive visualization, directly linking to parsed program code.

14.3.5 In-Depth Validation of *AUTHify*

Throughout this thesis, I evaluated the *RESTify* concern in significantly greater depth than the second concern candidate, *AUTHify*. Concentrating on *RESTify* is a straightforward choice, as *AUTHify* builds on top of the *RESTify* concern. But also, evaluating a second concern to this extent would have gone beyond the resources available for this thesis. Already conducting a single controlled experiment with *RESTify* required an advanced and notably sufficiently stable concern toolchain implementation. Also, running a second study, from recruitment to crafting the material, to the statistical analysis was not possible within the given time constraints of this thesis.
And yet, a second fully operational concern implementation, next to a thoroughly conducted second controlled experiment bears the potential to further validate the insights of this thesis. Future research could, the same way it has been performed with *RESTify*, apply the step-by-step concern integration projected in Chapter 7 as a second validation of the *FIDDLR* guidelines. Afterwards, this concern toolchain implementation could be used to perform

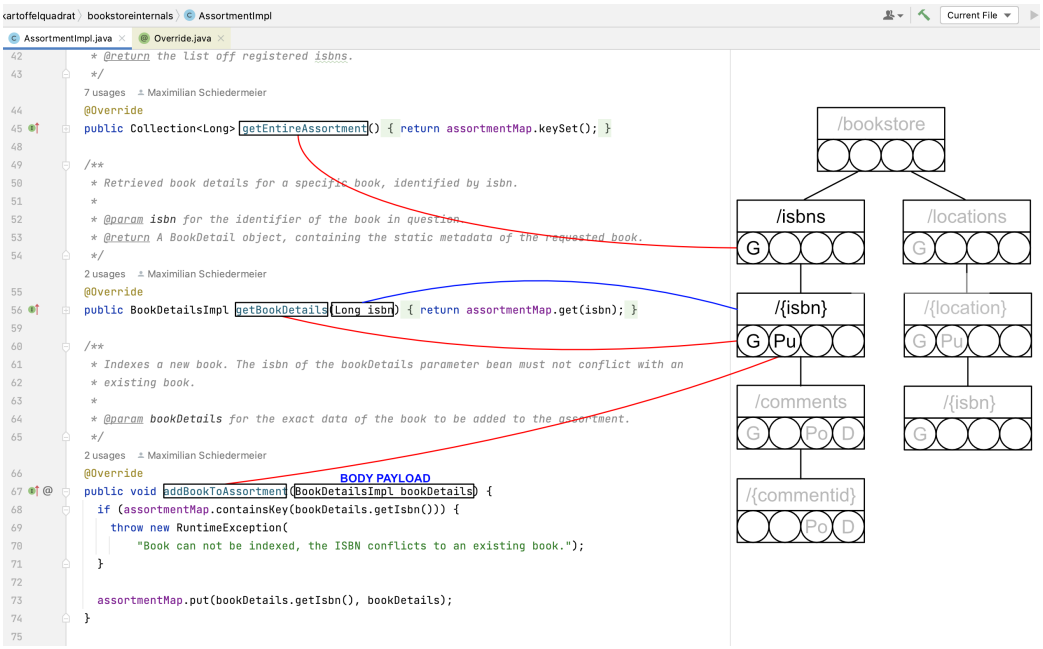

Figure 14.1: Conceptual Mockup of *ResTL* IntelliJ IDE Plugin

a second controlled experiment, following the same crossover layout as for *RESTify*. That is, participants could be hired to secure two sample applications for delegated access, compliant with the OAuth2 protocol. It would be interesting to compare the user feedback for this second concern to the one collected throughout the *RESTify* controlled experiment. Also, follow-up tests and time measurements could be analyzed to compare performances by methodology. It would be highly interesting to compare the outcome to the statistics presented in Chapter 12. Finally, in the spirit of the previous section, it would be likewise exciting to directly target an IDE plugin implementation of an *AUTHify* toolchain.

14.3.6 RESTify Error Categorization

Throughout the *RESTify* Controlled Experiment Chapter 9, we collected detailed data on the quality of individual participant submissions, i.e. we measured REST API correctness with unit tests. For the study, I was mainly interested in overall statistical trends, that is, whether a certain methodology would on average provide a significant advantage, regarding test pass rates. Also, we were more interested in the average submission qualities than in

the test outcome of individual tests. However, we still collected detailed test data. This data could be analyzed to gain deeper insights into the difficulty of individual REST endpoints. Figure 14.2 and Figure 14.3 illustrate how this could take place. These radar charts illustrate the average test pass rate on a per-endpoint basis. The four coloured contours describe the average group test success rates per endpoint. Contours lying further outside indicate higher pass rates and contours closer to the centre represent lower pass rates.

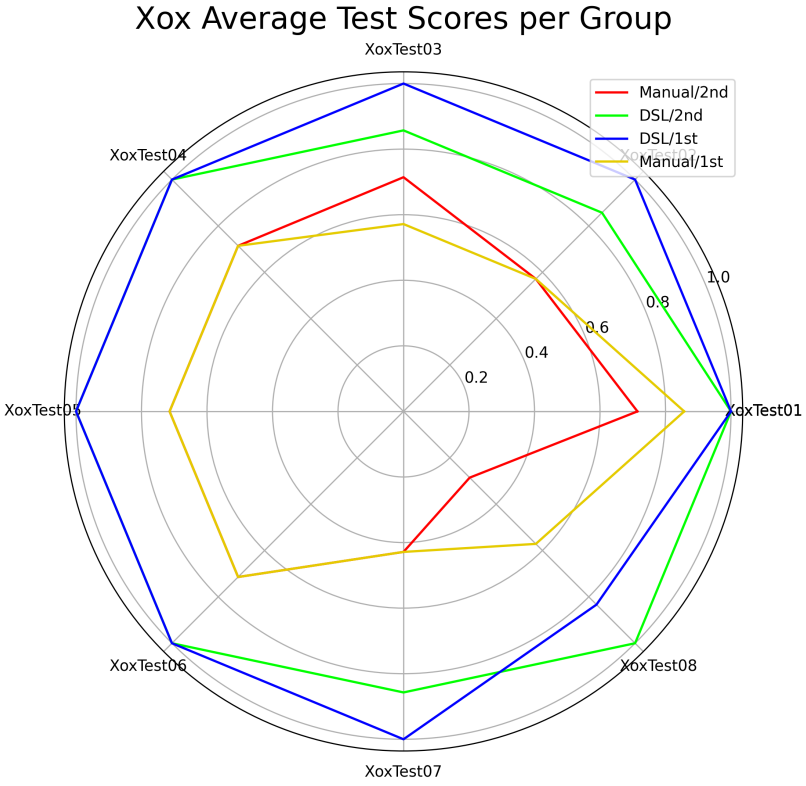

Figure 14.2: Radar Chart of Error Frequencies per REST Operation and Group, for the Xox Application

The statistical tests performed so far were only comparing the groups' overall performances, that is to say, whether contours on average were further to the outside or closer to the centre, in correlation to which methodology was applied.

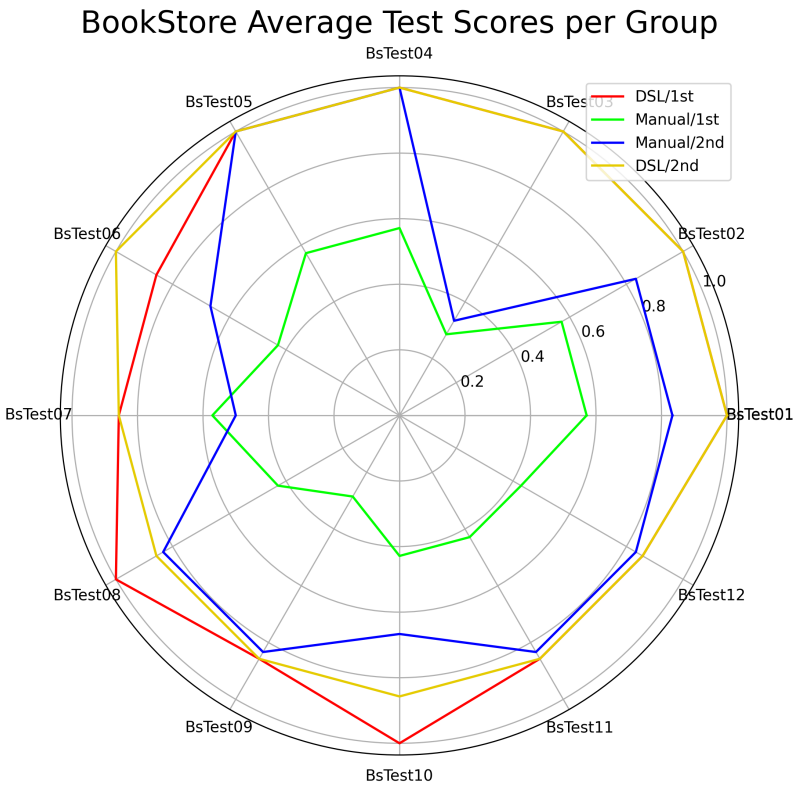

Figure 14.3: Radar Chart of Error Frequencies per REST Operation and Group, for the BookStore Application

We could also imagine investigating which individual tests showcase significantly higher or lower test pass rates. This translates to spikes in the contours, where results for an individual test are on average lower or higher than expected for a group. For instance `BsTest03` was significantly below the group average for the green and blue contours. Both were manual submissions. It would be interesting to inspect the code submissions for this endpoint and investigate if there is a common mistake that explains the lower pass rate. Similarly, the red and yellow groups performed unevenly well for different tests. In turn, it would be interesting to investigate if the advantages of the CSL methodology are dependent on the nature or complexity of specific REST operations. That is, it would allow us to investigate if

the general advantages associated with CSLs have a stronger effect on some specific engineering challenges.
A better understanding of endpoints that were statistically more error-prone could allow for the creation of improved guidance techniques, to offset common mistakes.

14.3.7 Revised Study

The *RESTify* Controlled Experiment was designed, performed and analyzed with diligence. However, as with every project, several essential insights occurred only throughout the study conduct. A revision of the study would certainly allow to offset several drawbacks.

Usually, empirical research is performed with a sample size estimation, that is to say, a preliminary calculation to estimate the minimum population size required, to run the intended statistical tests and analyze the collected data. Sample size estimations often require an educated guess on the expected data distributions, which can, e.g., stem from a pilot study. We did not perform a sample size estimation. This was mainly because of the limited funding. It was clear from the start that we should hire as many participants as possible with the available funding as a limiting factor.
The GLMs are an example of models that were on the verge of significance with the given sample sizes. Notably, the amount of sampling points prevented me from following some best practice recommendations for crossover experiments, notably the application of Linear Mixed Models (LMM), which internally perform multiple nested linear regressions and therefore require more sampling data. The study could significantly gain conclusiveness if performed with more participants, which in turn allows the use of LMMs.

Good practice for any controlled experiment is to only set control for either the experiment setup (i.e., the applications to work with), or the methodologies to assess (CSL concern) - but not both at the same time. The *RESTify* reference implementation allows conversion of applications to REST, however, technical intricacies cause some technical assumptions on the sample applications, for instance, that all functionality must be provided by singleton classes. This prevented us from performing the study on sample applications outside our control, e.g. sample applications created by other developers on GitHub.

In general, experiment instructions should be held in a way that favours neither experiment approach. Therefore we presented the expected target

REST interface textually, that is to say as human-readable text, describing the structure, from resource root to bottom. We decided on this presentation because testing against an interface is significantly easier than testing against the correct adaptation of a paradigm. However, technically we should have provided the instructions in a more abstract form, which would allow the participants to determine the target interface structure on their own. Instead of canonic unit testing, the submissions could have then been verified by a paradigm checker, that is a program that verifies if the submitted resource structure corresponds to common REST paradigms and if there is a way to correctly invoke the target methods. However, to the best of my knowledge, no such paradigm checker exists to date. It would be an exciting first step to develop a paradigm checker prototype and assess the submissions collected so far.

14.3.8 Testing Pathological Inputs with Fuzzing

Related to the previous point is the exploration of more advanced ways of testing RESTful services. As mentioned before, testing in the context of the *RESTify* study was straightforward, for we stipulated a target interface and knew exactly which test requests to send to the participant submissions. However, testing of valid inputs alone is insufficient. Sophisticated testing must also verify the handling of faulty requests or in the worst case pathological tests. In the context of REST, pathological tests correspond to inputs that cause internal server errors, server crashes, or high resource consumption that reduces responsiveness. A revised repetition of the *RESTify* controlled experiment should, notably if the target API is not stipulated, include advanced testing mechanisms that only verify the correct handling of sane inputs, but also test handling of pathological inputs.

Commonly these inputs are determined by two orthogonal approaches: *symbolic testing*, which is a formal approach that requires knowledge of the testing target, or *fuzzing*, which bombards the testing target with mutations of sane inputs [LPSS18]. In this context, fuzzing is the more promising approach, as it does not require knowledge of the testing target internals, and is in general considered more effective in detecting implementation issues than symbolic execution.

Most fuzzing frameworks are built for classic software libraries, written in C/C++ or Java. However, there are fuzzing frameworks specifically for REST [AGP19]. Once more, the already collected submissions could be tested concerning their robustness against pathological inputs. Note that

the RESTification process likely adds new vulnerabilities, notably due to implicit parameter conversions and en/decodings, and a generally higher resource consumption per API call due to the integrated web-server stack.

14.3.9 Reusable Crossover Experiment Suite

Empirical research is, just like software engineering, subject to a plethora of pitfalls, but also best practices. In my case, the correct conduct and design of the *RESTify* controlled experiment required thorough familiarization with existing literature. Notably, [VAJ16] was an insightful reference. Hence comes the idea of supporting the conduct of certain experiment types with reusable software models, similar to concerns.
Throughout the *RESTify* experiment, I set on a fully programmatic approach, that is to say, the entire experiment analysis can be replicated with the execution of a structured script, which executes one statistical test after the other. All tests and figure generators were written in a single, well-structured Python project, which was a reasonable choice for data analysis purposes [Sch22b]. Originally the motivation for this approach was also a strong interest in a sound replication package, which would allow any other research team to validate our data and analysis with minimal effort.
However, the more I developed with reuse in mind, I noticed that most of my modules would qualify for other crossover experiments, either with none or only minimal code changes.

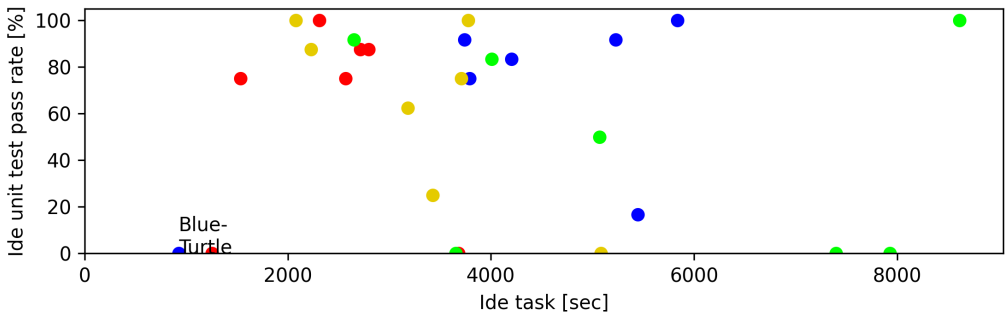

Figure 14.4: Blue Turtle (Sampling Point in Bottom-Left Corner was Identified as Scammer). They Spent the Least Time, Not a Single Test Passed. Further Indicators Confirmed Suspicion

At some point, I even pushed things to the extreme. The program had a clear categorization of all measured parameters, which allowed the decla-

ration of a simple loop to combine any two metrics and create a scatter plot with all participants. Note that in contrast to "*fishing for results*" (massive application of statistical tests on random data subsets, until some test happens to reflect the wished-for results), data exploration, especially visual techniques, is explicitly considered good practice for empirical software experiments [KPP+02]. This turned out to be useful, as it allowed the detection of an outlier in the scatter with "least time spent" and "worst test results", as shown in Figure 14.4.

Although the analyzer project still showcases many implementation details specific to our experiment, it would be an interesting enterprise to attempt to elaborate a general framework for crossover-controlled experiments. This could not only greatly speed up the creation of further, sound replication packages, but also serve as a best practice guide, with a set of modules tailored for this experiment category. Such a code refactoring project would by itself constitute an interesting case study, and in the best case be beneficial to the empirical research community, notably since to the best of my knowledge no such tool exists.

Acronyms

List of acronyms used in this thesis:

- **AOM**: ***A****spect-****O****riented* ***M****odelling*
- **AOP**: ***A****spect-****O****riented* ***P****rogramming*
- **AS**: *(OAuth2)* ***A****uthorization* ***S****erver*
- **CI**: *(CORE)* ***C****ustomization* ***I****nterface*
- **CORE**: ***C****oncern-****O****riented* ***RE****use*
- **CSL / CSML**: ***C****oncern-****S****pecific (****M****odelling)* ***L****anguage*
- **DevOps**: ***Dev****elopment and* ***Op****eration****s***
- **DSL / DSML**: ***D****omain-****S****pecific (****M****odelling)* ***L****anguage*
- **FIDDLR**: ***F****ramework for the* ***I****ntegration of* ***D****omain-Specific Mo****D****elling* ***L****anguages with Concern-Oriented* ***R****euse*
- **GLM**: ***G****eneral* ***L****inear* ***M****odel*
- **GPL / GPML**: ***G****eneral-****P****urpose (****M****odelling)* ***L****anguage*
- **GUI**: ***G****raphical* ***U****ser* ***I****nterface*
- **LEM**: ***L****anguage* ***E****lement* ***M****apping*
- **LMM**: ***L****inear* ***M****ixed* ***M****odel*
- **MDE**: ***M****odel-****D****riven* ***E****ngineering*
- **MSA**: ***M****icro-****S****ervice* ***A****rchitecture*

- **OAS**: ***O**pen**A**PI / **S**wagger*
- **PoC**: ***P**roof **o**f **C**oncept*
- **SoC**: ***S**eparation **o**f **C**oncerns*
- **RAML**: ***R**ESTful **A**PI **M**odelling **L**anguage*
- **RBAC**: ***R**ole **B**ased **A**ccess **C**ontrol*
- **REST**: ***RE**presentational **S**tate **T**ransfer*
- **RO**: *(OAuth2) **R**esource **O**wner*
- **RPC**: ***R**emote **P**rocedure **C**all*
- **RS**: *(OAuth2) **R**esource **S**erver*
- **SLOC**: ***S**ource **L**ines **O**f **C**ode*
- **SOAP**: ***S**imple **O**bject **A**ccess **P**rotocol*
- **SpEL**: ***Sp**ring **E**xpression **L**anguage*
- **UI**: *(CORE) **U**sage **I**nterface*
- **UML**: ***U**nified **M**odelling **L**anguage*
- **VI**: *(CORE) **V**ariation **I**nterface*
- **WADL**: ***W**eb **A**pplication **D**escription **L**anguage*
- **WRML**: ***W**eb **R**esource **M**odelling **L**anguage*

Bibliography

[AGP19] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. RESTler: Stateful REST API Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758, May 2019. ISSN: 1558-1225.

[AK08] Colin Atkinson and Thomas Kühne. Reducing accidental complexity in domain models. *Software & Systems Modeling*, 7(3):345–359, 2008.

[AKM13] Omar Alam, Jörg Kienzle, and Gunter Mussbacher. Concern-oriented software design. In *Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems - MODELS 2013*, volume 8107 of *Lecture Notes in Computer Science*, pages 604–621, Berlin, Heidelberg, 2013. Springer.

[AMK22] Hyacinth Ali, Gunter Mussbacher, and Jörg Kienzle. Perspectives to promote modularity, reusability, and consistency in multi-language systems. *Innovations in Systems and Software Engineering*, 18(1):5–37, March 2022.

[AZKA11] Masoom Alam, Xinwen Zhang, Kamran Khan, and Gohar Ali. xDAuth: a scalable and lightweight framework for cross domain access control and delegation. In *Proceedings of the 16th ACM symposium on Access control models and technologies*, pages 31–40, Innsbruck Austria, June 2011. ACM.

[Bae21] Baeldung. Spring Security OAuth Authorization Server, March 2021. https://www.baeldung.com/spring-security-oauth-auth-server.

[BFG10] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Journal of Computer Security*, 18(4):619–665, June 2010.

[Bum23] Bumble. Bumble - Share More With Bumble's Instagram Integration | Bumble, November 2023. https://bumble.com/the-buzz/bumble-profile-instagram-integration.

[Bur09] Bill Burke. *RESTful Java with JAX-RS*. O'Reilly, 2009.

[Car23] Aviad Carmel. Oh-Auth - Abusing OAuth to take over millions of accounts, 2023. https://salt.security/blog/oh-auth-abusing-oauth-to-take-over-millions-of-accounts.

[CCS01] Thomas Cook, Donald Campbell, and William Shadish. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Wadsworth Publishing, 2001.

[CDPF+13] Mariano Ceccato, Massimiliano Di Penta, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empirical Software Engineering*, February 2013.

[Cir23] CircleCI. GitHub OAuth app integration overview - CircleCI, November 2023. https://circleci.com/docs/github-integration/.

[CMH19] Cristina Cachero, Santiago Meliá, and Jesús M. Hermida. Impact of model notations on the productivity of domain modelling: An empirical study. *Information and Software Technology*, 108:78–87, April 2019.

[CRA23] Scotia Bank Canada Revenue Agency. CRA Direct Deposits, November 2023. https://www.scotiabank.com/content/scotiabank/ca/en/personal/bank-your-way/digital-banking-guide/banking-basics/cra-direct-deposits.html.

[CS11] Donald T. Campbell and Julian C. Stanley. *Experimental and quasi-experimental designs for research*. Wadsworth, Belomt, CA, 2011.

[DEPC21] África Domingo, Jorge Echeverría, Óscar Pastor, and Carlos Cetina. Comparing uml-based and dsl-based modeling from subjective and objective perspectives. In Marcello La Rosa, Shazia Sadiq, and Ernest Teniente, editors, *Advanced Information Systems Engineering*, pages 483–498, Cham, 2021. Springer International Publishing.

[DGL+17] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: Yesterday, Today, and Tomorrow. In Manuel Mazzara and Bertrand Meyer, editors, *Present and Ulterior Software Engineering*, pages 195–216. Springer International Publishing, Cham, 2017.

[Dij76] Edsger Wybe Dijkstra. *A discipline of programming*, volume 1. Prentice-Hall, Hoboken, NJ, 1976.

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network -based software architectures*. Ph.D., University of California, Irvine, United States – California, 2000. ISBN: 9780599871182.

[For12] Internet Engineering Task Force. OAuth2 Protocol Specification. Request for Comments RFC 6749, Internet Engineering Task Force, https://datatracker.ietf.org/doc/rfc6749, October 2012. Num Pages: 76.

[Fou21a] Apache Software Foundation. Apache CXF Documentation. http://cxf.apache.org/docs/jax-rs.html, 2021.

[Fou21b] Eclipse Foundation. Eclipse Jersey User Guide. https://eclipse-ee4j.github.io/jersey/, 2021.

[Fou21c] Eclipse Foundation. Jakarta restful webservices online 3.0 specification. https://jakarta.ee/specifications/restful-ws/3.0/jakarta-restful-ws-spec-3.0.html, 2021.

[Fow10] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010.

[FRBS04] Eric Freeman, Elisabeth Robson, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly, 2004.

[FS15] Marios Fokaefs and Eleni Stroulia. Using wadl specifications to develop and maintain rest client applications. In *2015 IEEE International Conference on Web Services*, pages 81–88, Piscataway, NJ, USA, 2015. IEEE.

[FTE+17] Roy T. Fielding, Richard N. Taylor, Justin R. Erenkrantz, Michael M. Gorlick, Jim Whitehead, Rohit Khare, and Peyman Oreizy. Reflections on the REST architectural style and "principled design of the modern web architecture" (impact paper award). In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 4–14, New York, NY, USA, August 2017. Association for Computing Machinery.

[FZB+18] Robert Feldt, Thomas Zimmermann, Gunnar R. Bergersen, Davide Falessi, Andreas Jedlitschka, Natalia Juristo, Jürgen Münch, Markku Oivo, Per Runeson, Martin Shepperd, Dag I. K. Sjøberg, and Burak Turhan. Four commentaries on the use of students and professionals in empirical software engineering experiments. *Empirical Software Engineering*, 23(6):3801–3820, December 2018.

[GFC+08] Jeff Gray, Kathleen Fisher, Charles Consel, Gabor Karsai, Marjan Mernik, and Juha-Pekka Tolvanen. Dsls: The good, the bad, and the ugly. In *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA Companion '08, page 791–794, New York, NY, USA, 2008. Association for Computing Machinery.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, USA, 1995.

[Git23] GitHub. Scopes for OAuth apps, November 2023. https://ghdocs-prod.azurewebsites.net/en/apps/oauth-apps/building-oauth-apps/scopes-for-oauth-apps.

[Gra07] Jeff Gray. Domain-Specific Modeling. *Handbook of dynamic system modeling*, 7(Handbook of dynamic system modeling), 2007.

[Gul23] Robin Guldener. Why is OAuth still hard in 2023?, February 2023. https://www.nango.dev/blog/why-is-oauth-still-hard.

[Hat23] Red Hat. JBoss RESTEasy Specification, 2023.

[HWR14] John Hutchinson, Jon Whittle, and Mark Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming*, 89:144–161, September 2014.

[HWRK11] John Hutchinson, Jon Wittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical Assessment of MDE in Industry. 33rd International Conference on Software Engineering (ICSE 2011), Piscataway, NJ, 2011. IEEE.

[Jam08] M. Jamshidi. *System of systems engineering? New challenges for the 21st century*. Wiley, Hoboken, NJ, October 2008.

[JBo21] Red Hat / JBoss. JBoss RESTEasy JAX-RS Community DocBook and Javadoc Documentation. https://resteasy.github.io/docs/, 2021.

[JHL+21] Vinícius Julião, Alexander Holmquist, Flávio Lúcio, Celso Simões, and Fernando Pereira. Hapi: A Domain-Specific Language for the Declaration of Access Policies. In *25th Brazilian Symposium on Programming Languages*, pages 9–16, Joinville Brazil, September 2021. ACM.

[Joh24] Jeff Johnson. *Designing with the Mind in Mind :: UXmatters*. Morgan Kaufmann, 2024.

[JR17] Antonio Sanso Justin Richer. *OAuth 2 in Action*. Manning, 2017.

[Kan03] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison Wesley, second edition edition, 2003.

[Ken02] Stuart Kent. Model Driven Engineering. In Michael Butler, Luigia Petre, and Kaisa Sere, editors, *Integrated Formal Methods*, Lecture Notes in Computer Science, pages 286–298, Berlin, Heidelberg, 2002. Springer.

[KGCM18] Tomaž Kosar, Sašo Gaberc, Jeffrey C. Carver, and Marjan Mernik. Program comprehension of domain-specific and general-purpose languages: replication of a family of experiments using integrated development environments. *Empirical Software Engineering*, 23(5):2734–2763, 2018.

[Kie23] Jörg Kienzle. TouchCORE, 2023.

[KLB15] Amy J. Ko, Thomas D. LaToza, and Margaret M. Burnett. A practical guide to controlled experiments of software engineering tools with human participants. *Empir. Softw. Eng.*, 20(1):110–141, 2015.

[KMA+16] Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Belloir, Philippe Collet, Benoit Combemale, Julien Deantoni, Jacques Klein, and Bernhard Rumpe. VCU: the three dimensions of reuse. In *International Conference on Software Reuse*, pages 122–137, Berlin, Heidelberg, 2016. Springer.

[KMB+96] Richard B. Kieburtz, Laura McKinney, Jeffrey M. Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino P. Oliva, Tim Sheard, Ira Smith, and Lisa Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th International Conference on Software Engineering*, ICSE '96, page 542–552, USA, 1996. IEEE Computer Society.

[KPP+02] B.A. Kitchenham, S.L. Pfleeger, L.M. Pickard, P.W. Jones, D.C. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, August 2002.

[Kru92] Krueger. Software reuse. *CSURV: Computing Surveys*, 24:131–183, 1992.

[Lab21] SCORE Labs. TouchCORE user guide. http://touchcore.cs.mcgill.ca/, 2021. Accessed: 2021-09-24.

[LPSS18] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. PerfFuzz: automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 254–265, Amsterdam Netherlands, July 2018. ACM.

[Mas11] Mark Masse. *REST API Design Rulebook*. O'Reilly Media, 2011.

[Mas23] Mark Masse. WRML Language Definition, February 2023. https://github.com/wrml/wrml.

[MB18] Simon Maple and Andrew Binstock. JVM Ecosystem Report 2018 - About your Platform and Application, October 2018. https://snyk.io/blog/jvm-ecosystem-report-2018-platform-application/.

[MCM14] Yulkeidi Martínez, Cristina Cachero, and Santiago Meliá. Empirical study on the maintainability of Web applications: Model-driven Engineering vs Code-centric. *Empirical Software Engineering*, 19(6):1887–1920, December 2014.

[Mul21] MuleSoft. RESTful API Modeling Language definition. `https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/`, 2021. Accessed: 2021-04-28.

[Mul23] MuleSoft. Advanced REST Client, November 2023. https://www.advancedrestclient.com/.

[MV10] Raphael Mannadiar and Hans Vangheluwe. Modular synthesis of mobile device applications from domain-specific models. In *Proceedings of the 7th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, MOMPES '10, page 21–28, New York, NY, USA, 2010. Association for Computing Machinery.

[Obs23] Obscurify. Obscurify, November 2023. https://www.obscurifymusic.com/about.

[Okt23] Okta. JWT Decoder, 2023. https://jwt.io/.

[OMG17] OMG. Uml superstructure, v2.5.1. OMG Specification, 12 2017.

[Pos23a] Postman. Postman API Platform | Sign Up for Free, November 2023. https://www.postman.com.

[Pos23b] API Platform Postman. 2023 State of the API Report | API Technologies, September 2023. https://www.postman.com/state-of-api/api-technologies/.

[Pur04] Gregor N. Purdy. *Linux iptables: pocket reference*. O'Reilly, Sebastopol, CA, 2004.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[RR08] Leonard Richardson and Sam Ruby. *RESTful web services*. "O'Reilly Media, Inc.", Sebastopol, CA, 2008.

[Sch06] Douglas C Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2), 2006.

[Sch20] Maximilian Schiedermeier. A concern-oriented software engineering methodology for micro-service architectures. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, pages 1–5, New York, NY, USA, October 2020. Association for Computing Machinery.

[Sch21] Maximilian Schiedermeier. Book Store, October 2021. https://github.com/m5c/BookStoreInternals.

[Sch22a] Maximilian Schiedermeier. Pushing the boundaries of planned reuse with concern specific modelling languages. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '22, pages 229–232, New York, NY, USA, November 2022. Association for Computing Machinery.

[Sch22b] Maximilian Schiedermeier. RESTify Data Analysis, September 2022. https://github.com/m5c/RestifyJupyter.

[Sch22c] Maximilian Schiedermeier. Xox, February 2022. https://github.com/m5c/XoxInternals.

[Sch23a] Maximilian Schiedermeier. Book Store (Manually RESTified), January 2023. https://github.com/m5c/BookStoreManuallyRestified.

[Sch23b] Maximilian Schiedermeier. The Horsemen of Empirical Research Apocalypse, June 2023. https://www.cs.mcgill.ca/~mschie3/assets/thera-poster.pdf.

[Sch23c] Maximilian Schiedermeier. Restify Experiment Replication Package, November 2023. https://m5c.github.io/ExperimentReplicationPackage/.

[Sch23d] Maximilian Schiedermeier. Spring Security OAuth2 Samples, July 2023. https://github.com/m5c/spring-security-oauth.

[SGD+14] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K. Rajamani, Janice Tsai, and Jeannette M. Wing. Bootstrapping Privacy Compliance in Big Data Systems. In *2014 IEEE Symposium on Security and Privacy*, pages 327–342, San Jose, CA, May 2014. IEEE.

[SKK21] Maximilian Schiedermeier, Jörg Kienzle, and Bettina Kemme. FIDDLR: streamlining reuse with concern-specific modelling languages. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2021, pages 164–176, New York, NY, USA, November 2021. Association for Computing Machinery.

[SKK24] Maximilian Schiedermeier, Bettina Kemme, and Jörg Kienzle. Give me some REST: A Controlled Experiment to Study Domain-Specific Language Effects. In *MODELS*, Linz, Austria, 2024. SUBMITTED.

[SLL+21] Maximilian Schiedermeier, Bowen Li, Ryan Languay, Greta Freitag, Qiutan Wu, Jörg Kienzle, Hyacinth Ali, Ian Gauthier, and

Gunter Mussbacher. Multi-Language Support in TouchCORE. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 625–629, October 2021.

[Sma23] SmartBear. OpenAPI Specification - Version 3.0.3 | Swagger, November 2023. https://swagger.io/specification/.

[Spo23] Spotify. Client Credentials Flow | Spotify for Developers, October 2023. https://developer.spotify.com/documentation/web-api/tutorials/client-credentials-flow.

[Swa23] Swagger. API Code & Client Generator | Swagger Codegen, November 2023. https://swagger.io/tools/swagger-codegen/.

[TK19] Juha-Pekka Tolvanen and Steven Kelly. How domain-specific modeling languages address variability in product line development: Investigation of 23 cases. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*, SPLC '19, page 155–163, New York, NY, USA, 2019. Association for Computing Machinery.

[VAJ16] Sira Vegas, Cecilia Apa, and Natalia Juristo. Crossover Designs in Software Engineering Experiments: Benefits and Perils. *IEEE Transactions on Software Engineering*, 42(2):120–135, February 2016. Conference Name: IEEE Transactions on Software Engineering.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, jun 2000.

[vKES+18] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. Teastore: A micro-service reference application for benchmarking, modeling and resource management research. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 223–236, Piscataway, NJ, USA, 2018. IEEE.

[Wal22] Craig Walls. *Spring in Action*. Manning, 2022.

[WHH03] Claes Wohlin, Martin Höst, and Kennet Henningsson. Empirical Research Methods in Software Engineering. In Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Reidar Conradi, and Alf Inge Wang, editors, *Empirical Methods and Studies in Software Engineering*, volume 2765, pages 7–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. Series Title: Lecture Notes in Computer Science.

[Wil03] D. Wile. Lessons learned from real dsl experiments. In *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*, pages 10 pp.–, 2003.

[WRH+12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.