Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

Aspect-driven Development of Enterprise Information Systems

Karel Čemus

A Dissertation Thesis Submitted for
the Degree of Doctor of Philosophy

PhD Programme: Electrical Engineering and Information Technology

Branch of Study: Information Science and Computer Engineering

February 2019

Thesis Supervisor:

Doc. Ing. Jiří Vokřínek, Ph.D.
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo náměstí 13, 121 35 Praha 2
Czech Republic

Thesis Co-Supervisor:

Ing. Tomáš Černý, Ph.D.
Department of Computer Science
School of Engineering and Computer Science
Baylor University
One Bear Place #97141
Waco, Texas, US, 76798

Abstract

Contemporary enterprise information systems put high demands on existing development approaches. While these systems implement growing number of business processes, they maintain consistency of persisted data, integrate remote services, and expose the domain to end users and other systems. Unfortunately, existing development approaches usually do not recognize business rules as a significant concern, although they are defined by a business domain and cross-cut throughout a whole system. Since they participate in input validation in the user interface, in preconditions of business processes, and in invariants in a persistent storage, there is no single focal point, which makes them difficult to encapsulate. In addition, they tangle together with other concerns participating in various components of a system. Since existing development approaches often fail to separate concerns, developers must manually linearize this multidimensional space into linear source code, which produces high amount of repetitions and code duplication. Subsequent maintenance of a system is highly error prone and requires significant efforts.

This thesis addresses the separation of concerns in enterprise information systems. Utilizing existing approaches, this thesis introduces the aspect-driven development approach to design of information systems. The approach focuses on decomposition, efficient representation, and context-aware runtime integration of concerns in order to avoid manual repetitions, remove code duplication, and subsequently reduce development and maintenance efforts. While utilization of domain-specific languages enables involvement of domain experts into development, complexity of business rules defines new requirements on their representation. Since the proposed approach is abstract and agnostic to used technology and architecture, this thesis elaborates in depth its implementation into design of common components of the layered architecture, and demonstrates separation of concerns in distributed environments. Validity of the approach is evaluated and discussed in conducted case studies, which suggest its benefits and limitations including its suitability for context-aware systems, easier system evolution, and efficient business rules management.

Keywords: Enterprise Information Systems, Separation of Concerns, Aspect-oriented Programming, Model-driven Development, Business Rules, Maintenance

Abstrakt

Vývoj podnikových informačních systémů klade vysoké požadavky na jejich návrh a architekturu. Kromě implementace rostoucího počtu byznys procesů jsou tyto systémy zodpovědné i za ukládání a konzistenci dat, integraci se vzdálenými službami a zpřístupnění byznys procesů koncovým uživatelům a dalším systémům. Bohužel, současný přístup k návrhu a architektuře systému obvykle nezohledňuje byznys pravidla, přestože jsou součástí validace vstupu v uživatelském rozhraní, definují předpoklady jednotlivých kroků byznys procesů, ale i invarianty v úložišti dat. Jelikož se byznys pravidla prolínají se celým systémem, současný návrh je nedokáže zapouzdřit na jednom místě. Mimo to, ostatní zájmy a součásti systému v různých komponentách interagují s byznys pravidly, což vede vývojáře k linearizaci vícerozměrného prostoru tvořeného vzájemně nezávislými zájmy a komponentami. Výsledkem této linearizace je velký počet duplicit ve zdrojovém kódu, což výrazně zvyšuje chybovost a pracnost údržby systému.

Tato dizertační práce se zabývá oddělením a automatickým znovupoužitím nezávislých zájmů a komponent informačních systémů a klade důraz na reprezentaci a kontextové znovupoužití byznys pravidel. S využitím existujících přístupů navrhuje nový aspektově-řízený vývoj informačních systémů zaměřený na dekompozici, efektivní reprezentaci a kontextové znovupoužití zachycených informací. Důsledkem je snížení množství manuálních duplicit ve zdrojovém kódu, snížení počtu opakování stejné informace a zjednodušení celkového vývoje a údržby systému. Ačkoliv využití doménově specifických jazyků umožňuje zapojení doménových expertů do vývoje, komplexita byznys pravidel klade nové požadavky na jejich reprezentaci. Dizertační práce demonstruje navržený přístup k návrhu informačních systémů jeho implementací do designu třívrstvé architektury a diskutuje jeho využití v distribuovaném prostředí, ačkoliv samotný přístup je nezávislý na architektuře i použitých technologiích. Výsledky prezentované případové studie zhodnocují validitu a vlastnosti přístupu včetně jeho výhod a nevýhod, mezi které patří podpora kontextově specifických systémů, snazší údržba a efektivnější správa byznys pravidel.

Klíčová slova: informační systémy, oddělení zájmů, aspektově-orientované programování, modelem-řízený vývoj, byznys pravidla, údržba systému

Acknowledgements

My sincere thanks and appreciation go to Tomáš Černý, who supervised me through my masters and doctoral studies. This thesis would never be written without his great support, constant encouragement, and mentoring. He made my life much richer, connected me with great people, and provided me with numerous opportunities to enjoy the research and get experience.

I would like to thank to Jeff Donahoo at Baylor University for his feedback, and for pushing my research to right direction. His efforts and insight greatly contributed to the quality and completeness of this thesis. I am grateful for the scholarship I was awarded by the Czech Technical University in Prague, which enabled my visit and studies at Baylor University in Waco, Texas.

I would like to express my gratitude to my supervisor Jiří Vokřínek for his patience and support during my work on this thesis; and to my students Filip Klimeš and Ondřej Kratochvíl for inspiring collaboration.

This research was supported by the Grant Agency of the Czech Technical University, grants No. SGS12/147/OHK3/2T/13 and No. SGS14/198/OHK3/3T/13.

Finally, my greatest thanks go to my family for support and endless patience.

Contents

List of Figures

List of Tables

List of Listings

Part I

Problem Statement

Objectives of the Thesis

State of the Art

Chapter 1

Introduction

Digitalization in IT and the expansion of the Internet make companies move their business processes and data to Enterprise Information Systems (EISs) in order to open their business to the world, make it available online, and improve efficiency of their processes [26]. This results in high expectations for EISs. These systems have to manage large volume of structured data, implement complex business processes, communicate with other systems, be efficient in both development and maintenance, highly scalable, available 24/7, robust, durable, fault tolerant, and reliable [48]. Although designs, frameworks, and best practices have been under major development for past decades, the overall complexity and permanent growth of requirements still produce new questions and challenges as the contemporary approaches reach their limits [26].

Both new and existing systems[1] are subject to permanent improvement in reaction to emerging requirements to adjust existing processes and implement new features. Unfortunately, this permanent evolution and limits of conventional approaches tend to disrupt the architecture and pollute the source code with repetitions and inconsistencies, which eventually results in a barely maintainable system, where each update is expensive and highly error-prone [43]. Fortunately, best-practices, design patterns, and enterprise architectures can be used to reduce risks and ease the development process.

The layered architecture[2] is one of widely used conventional designs [42] and is considered by several standards [28, 102][3]. It usually divides the system into three components shown in Figure 1.1. The lowest *data/data source* layer implements access

[1] Contemporary development teams apply agile software development, e.g., Scrum, Feature-driven development, and Rational Unified Process to improve risk management, focus on product, costs reduction, and rapid delivery [1]. Agile techniques are usually iterative and incremental, i.e., the team goes through the whole software life-cycle with a small part of the system in a single iteration. Each feature is analyzed, implemented, tested, delivered and the process starts over again. Thus also the new systems are basically existing systems with intensive development.

[2] The layered architecture is described in many sources often using slightly different terminology nevertheless with the same core idea. This thesis uses Martin Fowler's notation [42]. Alternative notations recognize, e.g., *persistence/data access*, *application/service/business*, and *web* layers.

[3] Although the Java EE specification does not directly mention any architecture, the terminology and the structure of many Java Specification Requests (JSRs) closely follows layered architecture.

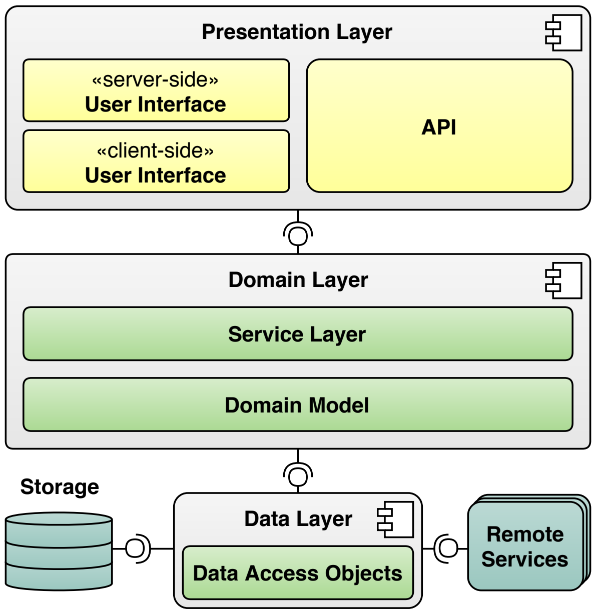


Figure 1.1: Layered architecture: Conventional design

to a persistent storage and remote services, e.g., a database or a file system in case of XML files. The middle *domain* layer encapsulates business logic, i.e., all business processes and business rules[4], in *services* inside this layer. Finally, the *presentation* layer exposes the system. Usually, there is a user interface (UI) often implemented as a web page and a programmatic API for communication with other systems.

The business processes implemented in the domain layer are divided into steps, each implementing a single transaction, i.e., it is located in a method of *service* [42]. Each service encapsulates all operations related to a single business purpose, e.g., the *Order Service* encapsulates methods related to the order creation, lookup, filtering, and updating. Such code organization makes steps repeatable, encapsulates the logic, and intuitively implements the business process flow.

Layered architecture, however, evinces some significant limitations with major impact on development and maintenance efforts. Reduced ability to reuse information already represented in source code could be considered among the greatest weaknesses [15]. There are two fundamental reasons. First, conventional, usually object-oriented, programming languages suffer from the inability to efficiently encapsulate cross-cutting concerns[5] [62]. Second, this architecture usually spreads through multiple technologies. For example, the data layer usually relies on SQL or similar language for querying relation databases or JSON for NoSQL databases. The domain layer

[4] Definition and understanding of business rules is essential for this thesis and is further elaborated on Chapter 2. For the purpose of this chapter, consider all constraints and rules attached to business processes to be business rules. For example, validation rules are considered as operation assumptions.

[5] Cross-cutting concerns are discussed in depth in Chapter 2. For the purpose of this chapter, consider these concerns for functionality to be considered in multiple places and cross-cutting through multiple components, methods, objects, and layers, e.g., logging or exception management.

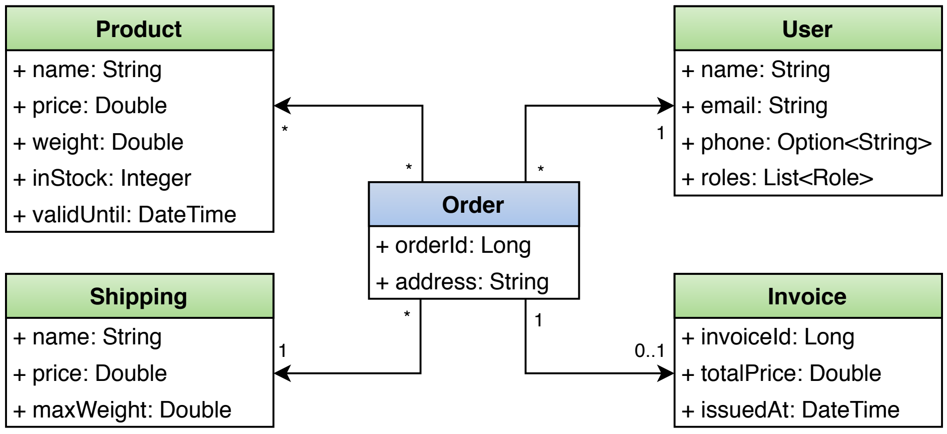


Figure 1.2: Example e-shop system: The class diagram

uses some general-purpose language such as Java or C# and the presentation layer often uses HTML, XML, or similar language. In consequence, with multiple languages and technologies in use, it is challenging to separate and reuse concerns scattered and spread throughout the whole system.

1.1 The Running Example

This work discusses the challenge of separation of concerns in multiple contexts, which is exacting to understand. Therefore, this section introduces a running example reused throughout the whole thesis for illustration purposes. It describes a simple information system, which is referenced in multiple chapters to demonstrate problems and make the discussion more readable and comprehensible.

A small e-commerce system will be used as an example[6]. Its data model is shown in Figure 1.2. The system maintains *Product*s for selling, and *User*s representing customers and employees. The system receives new *Order*s through its web-based graphical user interface and exposes its API for integration with other systems, e.g., an accounting system. Each *Order* maintains its internal list of ordered products, the selected *Shipping* carrier and the *Invoice*. The system recognizes three access roles: Ⓐ) a client buying products and creating orders, Ⓑ) staff responsible for the stock and the orders, and Ⓒ) an administrator maintaining the list of users and the product catalogue. Besides others, there are the following business rules:

① the name of a *Product* is not empty and at most 200 characters long;

② the price of a *Product* is positive;

③ the weight of a *Product* is not negative;

④ a *Product* can be created only by an administrator;

[6] The example is simplified and such a system would not be sufficient for real use. It is only intended to demonstrate and discuss particular challenges as a complex design would significantly impair comprehensibility.

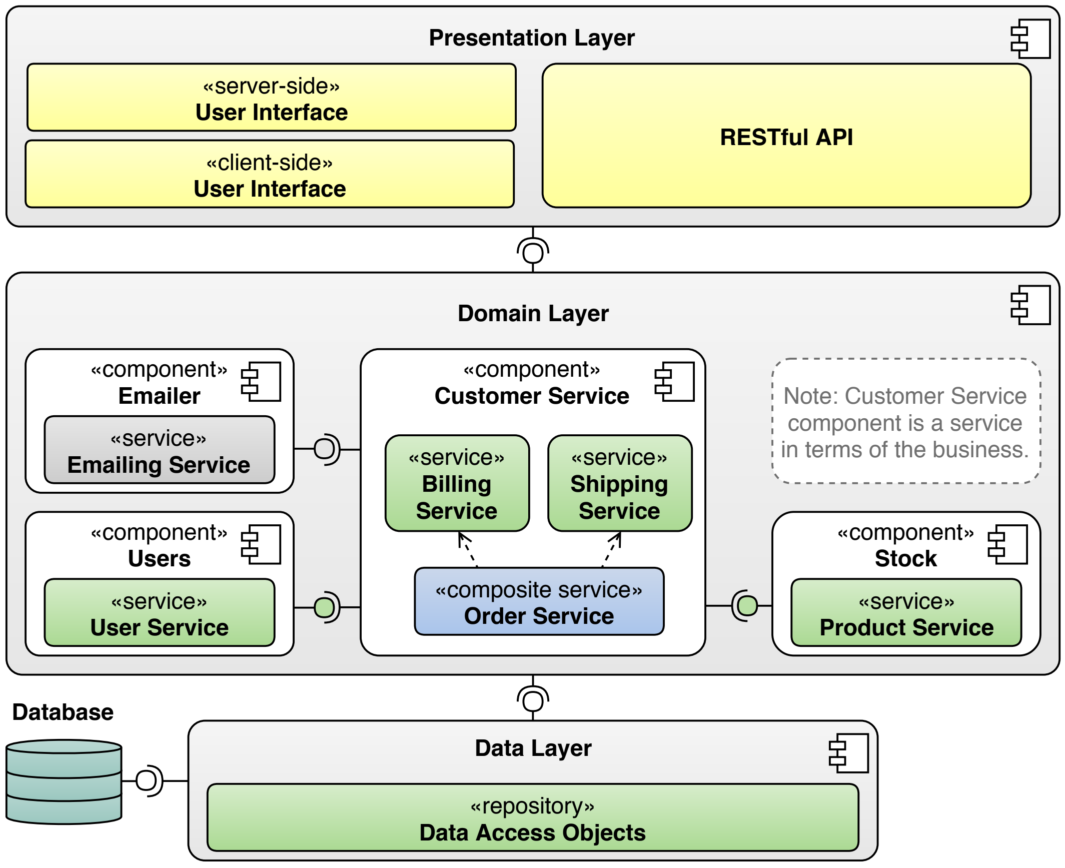

Presentation Layer
«server-side» User Interface
«client-side» User Interface
RESTful API
Domain Layer
«component» Emailer
«service» Emailing Service
«component» Customer Service
«service» Billing Service
«service» Shipping Service
«composite service» Order Service
Note: Customer Service component is a service in terms of the business.
«component» Users
«service» User Service
«component» Stock
«service» Product Service
Database
Data Layer
«repository» Data Access Objects

Figure 1.3: Example e-shop system: The architecture

⑤ a *Product* is available in the list of products (i.e. is not archived);
⑥ the name of a *User* is not empty and at most 50 characters long;
⑦ the email of a *User* is required, must be valid, and at most 200 characters long;
⑧ the list of products in an *Order* cannot be empty;
⑨ the invoiceId in an *Invoice* is unique.

The overall system architecture is shown in Figure 1.3. It follows the three-layered architecture with data access objects in the data layer. The domain layer is organized into four components, each responsible for a different part of the business. The most complex *OrderService* depends on all services as it aggregates information and delegates responsibilities. Finally, the presentation layer consists of a web-based graphical user interface and the RESTful API [36] exposing data to other systems. The API also makes the system ready for development of a mobile application.

This example illustrates the contemporary approach to design and development of EISs. Although it follows the best practices, standards, and the conventional architecture, there exist multiple challenges developers face to. In consequence, this example is reused later in this work to elaborate on those challenges and to illustrate an alternative approach addressing identified limitations of conventional development.

1.2 Problem Statement

Separation and representation of cross-cutting concerns is a significant challenge of conventional development, since each system considers many concerns, for example: a structure of the model, business rules such as validation rules, system localization, security policy such as an access control list, layouts in a user interface, widgets in a user interface, and others. While each concern has a representation and its scope is limited[7], existence of multiple concerns results in various problems. Usually, a representation comes with a specific language or a technology, and most concerns cross-cut throughout a system, its part, or with each other [A.3, 59, 105]. For example, while implementing a user interface, a developer considers a domain model, validation rules, security policy, a layout of the interface, and current localization. As a result, developers face to problems emerging from amount, nature, characteristic of cross-cutting concerns since conventional approaches lack the ability of separation of concerns. This inability results in the following critical problems, which are all addressed by this thesis.

Problem 1: Concern tangling and information repetition. While concerns are often orthogonal to each other and compose a multidimensional space, conventional approaches are unable to separate them in the linear source code. Consequently, the concerns are manually linearized, which results in significant information repetition and concerns tangling (Figure 1.4). These repetitions entail a high risk of introducing inconsistencies during maintenance, as it is easy to overlook some places to update, when they are not easily spotted and are distributed through the whole code [32, 43].

Consider a UI for a *Product* management from the example above, e.g., a view for update of a product. In such a view, there have to be considered the *Product* entity, its fields and data types, and validation constraints such as ①, ②, and ③ attached to the fields. Next, the view supports a layout for regular displays and an alternative layout for small mobile devices. Each field is rendered with a proper widget based on the field type, e.g., a string field uses a text, while a long text field uses a textarea element. Finally, all messages and labels are localized. As is illustrated in Figure 1.4, all these concerns are considered within a few lines of source code, which leads to significant code tangling and information repetition.

Problem 2: Limited synchronization of concerns over multiple technologies. Concerns cross-cut throughout the system and with each other, and are considered at multiple places in multiple layers and components. Since information systems utilize various programming languages and technologies, the concerns are not located in a single point but distributed throughout them. Subsequently, manual synchronization of the locations is highly error-prone, while automated reuse of concerns is limited.

[7] For example, layouts and widgets are often represented in XML or HTML and used only within the UI, but the rest of the presentation layer uses some general programming language such as Java.

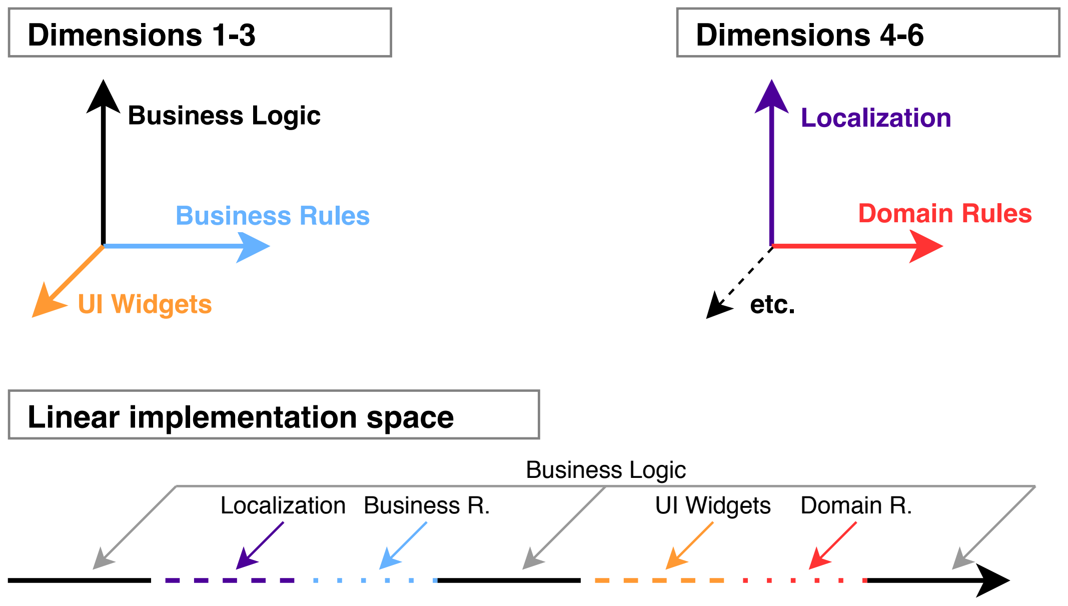


Figure 1.4: Orthogonal concerns in the multidimensional space

Consider a *Product* update scenario using the example above. There are validation rules such as ①, ②, and ③. All of them are considered in the following places, which results in multiple repetitions.

- First, the rules are used in the UI for client-side validation improving user experience, providing a modern user-friendly interface, and displaying proper user-friendly error messages.
- Second, the rules are considered for server-side validation inside the presentation layer to double-check the input.
- Third, the rules also apply in the RESTful API to properly validate the input and return an error message if necessary.
- Fourth, there is an input validation in the business operation in the domain layer, which actually updates the product.
- And finally, there are integrity constraints in the data layer protecting data consistency from corruption.

There are four languages in use. First, there is a general programming language, e.g., Java, implementing the domain layer and the server-side and RESTful API of the presentation layer. Second, there is a language such as SQL, JPQL, or HQL [6, 103] used to query a persistent storage in the data layer. Third, the client-side of the presentation layer utilizes JavaScript, and finally, HTML implements the UI and participates in the input validation. In consequence, since automated concerns reuse is significantly limited and considering the amount of repetitions (Problem 1), developers have to maintain and synchronize concerns distributed over all the languages and technologies, which significantly increases error-proneness and development efforts.

Problem 3: No centralization and no single point of truth. In general, developers intuitively encapsulate concerns and business logic in a single place. In case of a domain model and business logic, the object-oriented paradigm [67] uses objects to represent entities and to encapsulate the logic[8]. However, it fails to encapsulate the cross-cutting concerns [62], since there is no single point to locate the information. Such concerns are considered at many places within a system regardless the layer and the technology, which prevents developers from finding a single central place [62]. Consider the example from Problem 2. These business rules cross-cut the whole system but there is no single point where to concentrate locate them.

Unfortunately, the absence of the single point of truth, which would always indicate a desired behavior and configuration, significantly impacts evolution and maintenance of a project. Since the iterative and incremental development process and long-term project evolution require many changes in the system [1], project documentation often becomes obsolete and outdated [39]. Then, considering the significant code tangling (Problem 1) and multiple technologies in use (Problem 2), developers often face inconsistencies in the source code, which resulted from previous poor update. Furthermore, when the places are out of synchronization and there is no valid documentation, determining the truth and performing a correct update is challenging and error-prone.

Problem 4: Limited context-aware decisions. Consider the example in Problem 2, both administrators and staff are eligible for maintaining the products (④), but staff manages only current stock, while administrators maintain the whole product catalogue. When updating a product, the current context must be verified in all parts of the system (the UI, client-side validation, server-side validation, domain layer input validation), whether the current user is an administrator (Ⓒ) or only a staff member (Ⓑ) to properly secure the system and handle the request. Such a simple rule introduces at least four switches into the code, and each of them restates the rule itself.

While some approaches and frameworks partially encapsulate and reuse contextless business rules[9], context-aware rules such as the rules from the example above are beyond their limits. Such rules end up tangled and manually restated in the code even when a partial support is available. This limitation arises from a common understanding of conventional frameworks, which assumes contextless invocation, i.e., there is no consideration of user's context including his access role, geographical location, personal

[8] There exist multiple design patterns to encapsulate the logic. For example, there is the Rich Domain Model [42] encapsulating the logic inside a responsible object of the model. Contrary, the opposite Anemic Domain Model [40] with Transaction Script pattern [42] uses simple objects as a model, i.e., objects without any responsibility or functionality, and services implementing transactions, i.e., operations over the model. The layered architecture is usually used with the Transaction Script and the Anemic Domain Model.

[9] For example, Java Bean Validation [8] enables reuse of contextless business rules within Java. Nevertheless, exposure of the rules to other components and support of context-aware rules are missing.

settings, or the place of execution[10]. With modern systems, context-awareness becomes a significant requirement, which makes conventional approaches and frameworks are inefficient as they do not address context-aware decisions and concerns reuse [5].

Problem 5: High development and maintenance efforts. Essentially, all the problems defined above result in manual concerns tangling, information repetition, and source code duplication [67, 105]. The need of keeping all places synchronized jeopardizes the project because any inconsistency introduces a major security and business threat into the system and endangers system stability [43]. Subsequently, development process becomes demanding and requires thoroughness, efforts, extensive testing, and documentation. This might significantly increase the costs of the project.

In conclusion, conventional approaches to development and design of EISs suffer from limited support of cross-cutting concerns. High error-proneness, significant code tangling, and high development and maintenance efforts are direct results of these limitations. Therefore, this thesis addresses these limitations and proposes a novel development approach, which modifies design of EISs and treats cross-cutting concerns as first-class citizens of any system. Hence, it provides direct support for their representation, separation, and reuse to overcome these identified problems.

1.3 Objectives of the Thesis

Conventional approaches to design of enterprise information systems suffer from multiple problems identified and discussed in Section 1.2. Moreover, importance of business rules in EISs is not acknowledged, despite the rules being a significant concern as these systems implement number of business processes and maintain a business domain (Section 2.1). In consequence, limited support of separation of cross-cutting concerns in general results in their significant manual repetition in a system, and high development and maintenance efforts. Therefore, this thesis addresses the challenge of separation of concerns in enterprise information systems and sets the following objectives:

Objective 1: Define a novel approach addressing separation of concerns. Since conventional development approaches provide limited support of separation of concerns (Problem 2) and tend to manual concerns tangling and information repetition (Problem 1), define a novel approach to design and development of EISs addressing the above problems. Consider the layered architecture for illustration purposes.

[10] While the rule is always the same, the context is different. Even though both RESTful API and the client-side validate the input and produce a user-friendly error, the former uses server-side language and possibly returns some JSON object and HTTP status, while the latter updates the UI to inform the user. Next, the server-side validation and the validation of the input of the domain layer can throw an exception or stop the execution of request in some other way. Finally, the data layer applies the rule in the SQL language as an integrity constrain in a database.

Objective 2: Propose a mechanism reusing business rules in a system. Conventional representation of business rules lacks reusability. Propose a new mechanism utilizing business rules to reduce their manual repetition and enable automated transformation, distribution, and context-aware evaluation in a system (Problem 4). Implementation of the evaluation algorithm is not in the scope of this thesis.

Objective 3: Evaluate the impact of the approach on system design. Implement the approach into design of common components and discuss its utilization in a distributed environment. Evaluate its impact on separation of concerns, amount of repetitions in source code (Problem 3), and maintenance efforts (Problem 5). Then, evaluate efficiency of the approach, and discuss how the stated problems are addressed.

In conclusion, this thesis addresses the challenge of the separation of concerns in enterprise information systems focusing on reuse of business rules to reduce error-proneness, development and maintenance efforts, avoid problems described above, and increase overall efficiency of the development process.

1.4 Organization of the Thesis

Since this thesis deals with separation of concerns in enterprise information systems, the first part defines essential terms, approaches, and architectures in Chapter 2 to avoid ambiguity and misunderstanding. Chapter 3 discusses state of the art related to separation of concerns in EISs, more specifically, it elaborates both conventional and alternative approaches to EIS design, and discusses existing techniques and frameworks partially addressing the challenge. Since business rules belong among significant concerns in EISs, Chapter 3 also addresses their existing representations and evaluates their efficiency and suitability for the objectives of this thesis.

Second part contains the contribution of this thesis. Chapter 4 proposes the novel aspect-driven development approach, which refines contemporary design and development of information systems in order to address the objectives of this thesis. Following Chapter 5 discusses representation of business rules since their complexity makes separation of concerns more challenging. This chapter specifies requirements on the representation to implement the proposed approach and enable separation and reuse of business rules. Chapter 6 elaborates on the approach into design of multiple components and architectures of information systems, and discusses its impact on separation of concerns and subsequent maintenance.

Third part evaluates the proposed approach in Chapter 7, which discusses its efficiency using a case study conducted in order to compare the proposed and conventional approaches. The thesis concludes in Chapter 8, which highlights the contribution of this work, its impact on design and development of information systems, and summarizes challenges, set objectives, and how they were addressed.

Chapter 2

Basic Notation

This chapter clarifies and explains terminology used throughout this thesis to avoid any ambiguity. It summarizes essential aspects of fundamental approaches to ensure this work, its motivation, and contribution are easily comprehensible. All terms are presented without their impact on the topic, which is discussed later in this work.

2.1 Terminology: Business Logic

Understanding business logic terminology is essential for this thesis, because it deals with enterprise information systems and recognizes business rules as a significant concern. Therefore, as the proposed approach focuses on business processes and their operations, this section summarizes terms to avoid any misunderstanding.

Enterprise information systems (EISs) are enterprise applications tailored for a particular business domain to automate its processes and facilitate data maintenance [77, 106]. These systems often face various technological, design, and performance challenges due to volume of maintained data and complexity of business processes. Contemporary systems are often implemented as web applications, which expose their content to end users in a graphical user interface, and to other systems via web services or other communication protocols [67].

Business logic is part of a system encapsulating essential business processes defining creation, transformation, and flow of data. The logic consists of processes, operations, and procedures, is driven by business rules [77], and defines interactions among objects in a domain model. Implementation of business logic is imperative and usually operates on upper layers of a system as it considers the domain model, not the low-level concepts such as a database, protocols, and remote services. With constant business evolution and process optimization, business logic often changes [106].

A business process is a collection of coordinated activities and tasks to accomplish a greater business objective [116]. The objective must have business value, i.e., it is worth to a stakeholder. A business process consists of partial subtasks (business operations), incoming and outgoing events, and transition guards (business rules). Processes may be implemented both imperatively and declaratively, i.e., in a general-purpose language or using a domain-specific language such as BPMN [20], respectively. A business process is often visualized as a graph of activities together with its data flow.

Business rules are also known as domain rules[1] and define or constrain some aspects of business [51, 77]. Subsequently, they dictate how a domain or business operate [94]. The rules relate to real-life policies and include, e.g., company policies, physical laws, government laws, or domain rules, and pose as assumptions in a system [67]. Business rules can be written declaratively and represent invariants, preconditions, and postconditions based on the rule and the context of the execution [64, 73].

A business operation is a single step in a business process. It implements a subtask in the chain of activities to accomplish a greater objective [116]. Each operation consists of its *preconditions*, *invariants*, *postconditions* and a body. The body is the operation itself. Preconditions are business rules to be satisfied before the operation execution. When this assumption fails, the operation cannot be invoked. Invariants are business rules, which validity is not changed by the operation. Postconditions are business rules defining the result of the operation [64].

For example, consider the *Authenticate* operation taking a username and a password as its arguments. When these credentials are valid, the user is authenticated. There are preconditions: 1) the username is not empty, 2) the password is not empty, 3) the combination of the username and the password are valid credentials, which belong to a user. The invariant is the system state, which does not change, e.g., a user's shopping cart remain unchanged. The postconditions are: 1) there is non-empty user id in the context, 2) there is non-empty user's name in the context. In the body of the operation, the user is looked up by the username and assigned to the context [A.7].

This understanding of a business operation is often used with the event-driven rule-based system, where each operation is defined independently and the system listens to events. When an event occurs, preconditions of all operations are evaluated, and when they are satisfied, the body of the operation is invoked [11, 52].

In conclusion, business logic describes a business domain and defines how it works. Business processes describe activities and the data flow in the domain, and each process consists of business operations, which are small tasks composed together to achieve a greater objective. Each operation defines its assumptions and guaranties qualities of its results, which this thesis refers to as preconditions and postconditions, respectively.

[1] The term "domain rules" stands only for the domain-specific rules, i.e., the subset of all rules, which this thesis calls the "business rules". Unfortunately, there exist non-business-related systems such as military and weather-related, which are not supposed to have "business rules". In this context, "domain rules" are often understood as all rules regardless the ambiguity.

2.2 Terminology: Concerns

Conventional approaches to design and development of EISs usually suffer from limited and inefficient concerns management, which results in significant challenges and inefficiencies in project maintenance. This thesis proposes an approach recognizing some concerns to be a first-class citizens of EIS design to reduce these inefficiencies and improve quality of development. Therefore, it is essential to understand the terminology of concerns and related issues as this thesis frequently refers to it.

A concern is any part of a system, which a stakeholder considers as a conceptual unit [92]. Usually, a concern is identified on the basis of functionality and behavior. The concern is well-defined at the requirements level and at some level of abstraction, it stands either for a concrete behavior (a functional requirement) or a system-wide characteristic (a non-functional requirement) [76]. However, the clear boundaries of both these types of concerns fall apart at the code level. In source code, the concerns usually overlap, and all types of code units (a routine, a method, a function, an object, etc.) deal with multiple concerns at once [92]. Typical concerns in a software include features, nonfunctional requirements, design idioms, and implementation mechanisms.

Concern tangling is a degree to which the implementation of a concern intertwined with other concerns. It often raises from the difference in decomposition of requirements, models, and the code. While requirements are decomposed by a feature, the model is decomposed by an object and a component, and the code is decomposed by an object, a class, and an interface. Although concerns are originally isolated in the requirements level [76], this difference compromises comprehension and evolution of software and results in concerns tangling and manual code repetition [105].

The tangling has detrimental effect on the source code and arises from the poor separation of concerns. It often introduces code repetition and reduced comprehension. Modifying such code requires significant efforts and may produce a lot of defects [32, 43]. For example, implementation of a logging policy during the implementation of a security policy results in code tangling.

Concern scattering is another issue originating from a poor separation of concerns. A concern is scattered if it is referenced from multiple units instead of being isolated in a single location. For example, considering a transaction management, when each operation manages a transaction individually, then the concern and the related code are scattered throughout all those operations. The scattering usually goes along with concern tangling, code duplication, and loss of encapsulation and reusability [32, 105].

Cross-cutting concerns are concerns intertwining with other concerns. Usually, they have system-wide impact and refer to secondary requirements and thus stand for system-wide properties, while concerns representing a single and specific functionality for primary requirements, i.e., functional requirements, are known as core concerns [76]. The cross-cutting concerns cut through core concerns, must be considered in many modules and units of a system [62]. That makes them difficult to modularize and encapsulate, and the source code with overlapping concerns is difficult to read and maintain. Their cross-cutting nature often results in concern scattering and tangling with other concerns at the code level [92]. Typical representatives of cross-cutting concerns are logging and security policies.

Separation of concerns is well elaborated challenge connected to a decomposition and backward composition of concerns [105]. The need for the decomposition raises from the Single Responsibility Principle [70] and the Don't Repeat Yourself (DRY) principle [117]. The challenge lies in the nature of cross-cutting concerns, i.e., in their wide impact [105] and the mutual orthogonality in the multi-dimensional space.

The separation and encapsulation of self-standing concerns allows to keep the related information together and increase its cohesion and maintainability. The separation of concerns fights the concern scattering, because it is difficult for developers to locate and understand the code implementing a concern, when it is spread throughout the whole code base [92]. Although cross-cutting concerns cannot be clearly decomposed from the rest of a system by conventional techniques, there exist methods enabling the separation of cross-cutting concerns [62, 105] and their automated composition back together. It reduces manual concern tangling, increases modularity, maintainability, and subsequently reduces development costs and efforts.

Unfortunately, some concerns often cannot be decomposed and encapsulated [92]. This originates from an inadequate initial design, mutual concerns interaction in the N-dimensional space, or limitations of programming languages and tools [83]. In consequence, a full separation of concerns remains a challenge.

Single point of the truth sometimes referred to as a single focal point is a place with a full description of a concern. Such concentration of information significantly increases maintainability through highly cohesive information without any additionally tangled concerns. A single point of the truth encapsulates the fundamental truth and prevents ambiguity of inconsistencies. It also makes testing and concern validation through a review easier as there is no need to browse the whole code base.

With a properly defined distribution mechanism, concern description is distributed throughout the system without any need of manual information repetition, code duplication, or concern tangling. However, creation of such a point and a mechanism is difficult if even possible with conventional techniques, because there exist concerns, which are difficult to either encapsulate, or distribute, or both. Moreover, the idea of the single focal point hits the limits of the object-oriented programming and requires application of advanced approaches and methods [62, 105].

In conclusion, a concern is any part of a system considered as a unit by a stakeholder. Usually, it refers to a behavior or a functionality. Unfortunately, many concerns cross-cut throughout multiple layers, components, and technologies of a system, which results in significant challenges such as concern tangling, scattering, and absence of a single point of the truth as a multidimensional space collapses into linear source code. In consequence, it is important to maintain concerns separately in order to avoid inconsistencies and reduce maintenance efforts.

2.3 Terminology: Code Quality

Considering efficiency of development of an EIS, code quality is a significant aspect. It impacts error-proneness and both development and maintenance efforts. The approach proposed later in this thesis claims higher code quality delivered through the separation of concerns and more efficient concerns management. Therefore, this thesis utilizes code quality terminology and metrics to evaluate the approach and compare it to conventional approaches. As a result, this section summarizes and clarifies the most common terms in order to avoid any misunderstanding and unify the terminology.

Coupling is a metric indicating how elements are connected to, are aware of, and depend on other elements [69]. Strong coupling means highly connected elements, which results in difficult maintenance. Even a small change has a major impact and requires significant amount of work to modify the code, and such modification is highly error-prone due to its impact. Low coupling principle belongs among the cardinal rules and applies in multiple aspects of software development. Reducing the coupling means reducing the impact of a change, which includes reduction of time and efforts required to modify the system. The best practices to reduce the coupling and keep it low include design patterns [45], Demeter's Law, and GRASP patterns [67].

Cohesion is another metric often used for quality assurance. It indicates how strongly related and focused responsibilities of an element are. An element with strongly focused and related responsibilities has high cohesion, while an element doing many unrelated things or doing just too much of work has low cohesion and usually suffers from hard comprehension and difficult reuse and maintenance. Usually, low coupling and high cohesion go hand in hand and vice versa. There are various best practices to keep cohesion high such as design patterns [45] and GRASP patterns [67].

Code readability is a quality-defining metric considering the efforts required to understand the code. It is a part of overall comprehension, but requires human judgement, and thus belongs among subjective metrics. The best readability is achieved with a simple, focused, and cohesive code, with clearly defined responsibilities of objects, methods, and functions, and without unnecessary coupling among objects. Besides the complexity and structure, it also considers a naming strategy, i.e., efforts required to understand a purpose of a class, a method, and a variable [14]. There exist various best practices to maintain readable code, e.g., Keep It Simple, Stupid (KISS) principle, Don't Repeat Yourself (DRY) principle [117], and SOLID principle [70].

Code reuse refers to the possibility to use the code again somewhere else to avoid its duplication. This characteristic relates to the DRY principle, which suggests reusing the code instead of repeating it. The code duplication belongs among the most significant code smell techniques highly increasing maintenance efforts [43]. Reusable code is located in a single location, which minimizes maintenance efforts and helps avoid potential inconsistencies that could arise if multiple locations had to be modified.

Maintainability belongs among fundamental quality-defining metrics considering efforts required to maintain and modify the code. That includes both comprehension and actual modification. High maintainability assumes good code readability to ease its understanding, high cohesion to have clearly defined responsibilities, low coupling to reduce the structural complexity, proper design, and the absence of code duplication. Such code can be easily modified with minimal risk of introducing a defect or an inconsistency. Considering a project usually spends more than 70 % of its life-time in a maintenance phase [14], and that most of the time developers evolve existing code [55], then code maintainability significantly impacts overall costs of a project.

Error-proneness is a task quality metric indicating probability of making a defect. In this thesis, the error-proneness is usually discussed in relation to code or concern maintenance, which refers to its modification. Having code or a concern duplicated in multiple places means its modification very likely introduces a defect through an inconsistency, skipping some occurrence, or misunderstanding. The probability of making a mistake significantly raises with concern tangling and scattering. The best practices such as DRY and KISS principles, design and GRASP patterns, and keeping the code readable significantly reduce the likelihood of introducing a defect during maintenance.

Development and maintenance efforts encapsulate all tasks and activities performed during development and maintenance of a project, respectively. That includes analysis of requirements, their implementation, i.e., code understanding and evolution, then testing and delivery [67]. Even a small simplification of a routine task has a major impact on the overall efforts and costs considering the number of task executions. Such potential savings or expenses embrace the optimization of a process because repeatedly doing routine tasks without adding any value to given project mean pointless expenses. For example, maintenance of restated information, duplicated code, and efforts required to understand complex and poorly structured code are reducible by code refactoring to get better design and structure, and to get rid off code smell [43]. Use of tools and frameworks easing routine tasks is a big help to a project.

In conclusion, code characteristics clarified in this section help to evaluate and discuss quality of source code and subsequently compare quality of development approaches to design of a system. The terminology is utilized in Chapters 4 to 7 to discuss code organization and its impact on overall development and maintenance efforts.

2.4 Terminology: Approaches and Architectures

This thesis evaluates conventional development approaches and proposes an alternative approach to design of EISs. To prevent ambiguity, this section briefly summarizes fundamental background, while it is elaborated in detail in Chapter 3.

Layered architecture is a conventional architecture of EISs considered by various standards [28, 102], and this thesis uses the three-layered architecture for illustration purposes (Figure 1.1). While layered architecture addresses issues related to the separation of concerns and coupling and cohesion of business logic [42], the other concerns are not addressed at all and remain manually repeated instead.

Domain-specific languages are tailored to a concrete domain, efficiently capture its concerns, and store them aside runnable code [72]. The knowledge is usually managed by domain experts and benefits from changes implemented without a source code recompilation [41]. These languages appeared to be a valuable asset to enterprise applications due to more efficient domain description, comparing to general-purpose languages [54]. Correct deployment of a domain-specific language saves lots of efforts, contributes to separation of concerns, and increases code quality.

Object-oriented programming dominates system design and is based on intuitive mapping close to the real world. It recognizes objects with their attributes, templates (classes), and operations (methods) [67]. This mapping eases modelling of a domain and its subsequent transformation into a programming language. While it provides beneficial techniques such as encapsulation, polymorphism, and inheritance to reduce error-proneness and efforts [74], it suffers from inability to deal with cross-cutting concerns [104], which results in code duplication and concern tangling.

Aspect-oriented programming is a complementary paradigm to the object-oriented programming, evolved to encapsulate and reuse cross-cutting concerns [62]. It directly focuses on separation of concerns and their efficient domain-specific description [66]. Automated composition (weaving) of aspects positively impacts overall design and code qualities. It reduces coupling and increases cohesion through decomposition, while it reduces manual code duplication and information restatement [105].

Model-driven development maintains concerns in various models on different levels of abstraction, and applies transformation rules to produce more specific models and final source code [63]. While this forward transformation process facilitates concerns distribution and reuse, produced models must be manually refined and completed. In consequence, forward transformation significantly limits efficiency of the development, since each update of abstract models erases manual refinements in more specific models.

In conclusion, established understanding of these approaches helps communicate the challenge and the proposed approach elaborated in this thesis. Although Chapter 3 discusses approaches in greater detail and highlights their benefits and limitations, this general description is sufficient for understanding the rest of this thesis.

2.5 Summary

To avoid ambiguity and misunderstanding, this chapter briefly established terms considered throughout this thesis. It helps to fully grasp its contribution, and understand its motivation arising from limited support of the separation of concerns in conventional approaches. While business-related notation helps to communicate design and flow of information systems, classification of concerns is essential for understanding the motivation of this thesis. Then, summary of existing approaches defines necessary background knowledge for understanding the contribution and the approach proposed in Chapter 4. Finally, code quality notation enables discussion over quality of the approach, which is delivered in Chapters 6 and 7. In conclusion, while Chapter 3 elaborates this background in detail, this is the essential minimum for comprehension of the contribution of this thesis.

Chapter 3

State of the Art

Contemporary enterprise information systems deal with challenges arising from constantly growing volume of data, complexity of requirements, and size of business domain. This pressure significantly urges research of novel approaches and technologies as contemporary approaches reach their limits. This thesis addresses separation of concerns in information systems, which significantly impacts development and maintenance efforts, and error-proneness of change requests. This chapter summarizes the state of the art relevant for this thesis. Therefore, it introduces both conventional and alternative approaches to development of EISs and also elaborates their common architectures, which is further utilized in Chapter 4. Since business rules are a significant concern of these systems, this chapter evaluates existing business rules representations, which is later referred to in Chapter 5. Finally, this chapter discusses existing approaches to separation of concerns in common components of information systems, which is considered later in Chapter 6.

The running example from Chapter 1 and discussion of its problems in Section 1.2 show the importance of processes in EISs [106]. Consequently, business rules are a significant cross-cutting concern in a system [21, 62, 59], which has consequences such as concerns scattering, tangling, and limited reuse. Unfortunately, information systems also deal with other cross-cutting concerns, most of them is located in the UI [59]. In consequence, business rules belong among various cross-cutting concerns in a system, which often cause many problems (Problems 1 to 5). Therefore, this chapter elaborates contemporary approaches to the problems and the following chapters propose a new approach to design and development of information systems.

3.1 Common Architectures of Information Systems

Each software product including information systems has its own software architecture defining number of significant decisions related to the overall organization of a system [9]. The architecture has major impact on a project and determines efficiency of conducted tasks, since it defines organization of source code into logical modules, components, and layers [67]. While contemporary standards and mainstream technologies such as Java Enterprise Edition (Java EE) and .NET platform build systems using the layered architecture [28, 42, 67], alternative technologies often utilize the architectures based on the Model-View Separation principle. Both these architectures assume different underlying principles. While the layered architecture is based on the Anemic Domain Model [40, 85] and Transaction Script design pattern [42], architectures based on the Model-View Separation principle use Rich Domain Model with significantly different qualities. However, both architectures and design patterns partially deal with separation of cross-cutting concerns, which is discussed in this section.

3.1.1 Anemic vs. Rich Domain Model

Both conventional architectures differ mostly in the underlying domain models, which impacts the structure and qualities of a system, and directly define responsibilities and information distribution. The model represents a structure of a business domain, its objects and relations, which makes it an essential component of a system. Although any architecture can use any domain model, conventional designs mostly use specific combinations. While this section discusses two major approaches to implementation of a domain model, the next sections elaborate the architectures in detail.

First, the Rich Domain Model (RDM) [40] aligns the model with the structure of given business domain. It implements its processes, and prefers model completeness to avoid scattering and distribution of information outside the model [85]. Therefore, all concerns are already captured in the model or its dependencies, usually through highly-decorated classes and fields; e.g., business logic, database access, or field presentation widgets. Basically, each class closely relates to a business object and carries information on how to persist, validate, and render itself. Consequently, many fields and classes share a lot of configuration and mechanisms via inheritance and complex field data types to reduce the amount of source code, deliver clean and maintainable design, and avoid code duplication and information restatement.

Contrary, the Anemic Domain Model (ADM) [40] does not invest much efforts into development of a complex model. It replaces domain classes with data carriers, instead, which leads to Fat Service Layer and Transaction Script [42] design patterns accumulating business logic and related business rules in facade classes (services) [85]. However, this simplicity of the model fits to functional programming, which uses immutable data structures, pipelines, and transformations to implement simple and testable business logic [82]. Unfortunately, this distribution of concerns and responsibilities in information systems tends to significant concern repetition.

In conclusion, migration of business logic, business rules, and other concerns out of the domain model removes the single point of the truth and causes significant repetitions and concerns scattering, which tend to inconsistencies [15]. Although, the ADM leads to the Transaction Script pattern, which uses a less complex organization of business logic into data pipelines and transformations at the cost of low information encapsulation and high restatement. Contrary, the RDM deals with concerns through rich data types, which are maintained in the model in order to avoid their entangling into the rest of the system. Although, there exist context-specific concerns and concerns cross-cutting multiple operations at once, which hits the limitations of the RDM. It results in significantly increased complexity of the model as it does not provide any mechanism to express and reuse cross-cutting concerns, therefore it falls back to their repetition [85, 15]. To summarize the differences, while the ADM prefers simplicity over complexity at the cost of repetitions and increased efforts, the RDM prefers completeness over concerns tangling at the cost of higher complexity [85]. However, both approaches fail to address separation of concerns.

3.1.2 Layered Architecture

The layered architecture is one the of main architectures used in contemporary EISs [28, 102]. This use-case centric architecture focuses on organization of business logic and addresses issues related to the separation of concerns [67]. It deals with coupling and cohesion of application logic, its scattering throughout source code, and intertwining with more general code or code of the user interface. These efforts get along the DRY and KISS principles. Reduced complexity, increased reusability, and eased scalability belong among its significant benefits. Unfortunately, the separation of concerns is limited to application logic in the domain layer.

This architecture organizes the system into multiple layers, each encapsulating a single high-level concern to ensure its high cohesion and reduced coupling [42, 67]. Although the organization of the architecture may vary, it often consists of three layers or their variations, as is suggested in Figure 1.1 [42]. Despite clear definition of responsibilities of each layer, cross-cutting concerns cannot be encapsulated in any of them because they live outside the object-oriented programming and cross-cut throughout the whole system [62]. These concerns are considered in multiple locations and are orthogonal to this linear architecture, which makes them difficult to be captured in source code [105] as is elaborated in Section 3.3. For example, this includes auditing, security policy, and exception management.

Enterprise information systems implemented on the layered architecture suffer from two types of information repetition: horizontal and vertical [A.14]. The horizontal repetition is defined as a repetition inside of a single layer or a component, and thus occurs only inside a single technology and can be avoided by object-oriented principles such as encapsulation [43]. For instance, an input widget for a single-line string in the UI is repeated many times in a UI description, which makes it horizontally repeated. Contrary, the vertical repetition impacts multiple layers and often various technologies. For instance, consider the validation rules in the running example. They are considered in the data layer to protect data consistency, in the domain layer to validate assumptions of business operations, and in both the server-side and the client-side of the UI to validate the input. In consequence, information has various representations and uses, which makes reduction of vertical repetitions more challenging [A.14].

Another classification distinguishes repetition of semantics and references [A.14]. The repetition of semantics literally repeats the information itself, its meaning. For instance, a business rule repeats its implementation and how to verify it. Second, the repetition of references only duplicates addresses to another place with semantics definition. For instance of a business rule, there is repeated invocation of a validator, which implements verification of the rule [15]. Repetition of references is weaker and causes fewer issues as it decouples places of use from places of definition such as programming against a contract [67, 95]. This usually leads to information definition being repeated in less places, which results in better maintainability and easier to updating [43]. However, there still exists possibly high repetition of references depending on the used level of abstraction.

High information repetition in a system is caused by the architecture itself. This architecture usually comes with the Anemic Domain Model [85], which leads to use of the Fat Service Layer and the Transaction Script design patterns [42]. Unfortunately, while these patterns are suggested by several standards [28, 102], they break object-oriented principles and cause significant issues [42]. Despite simple transaction-oriented implementation of the domain layer, extraction of concerns out of a domain model leads to their significant repetitions in the rest of a system, which results from the absence of the single focal point. Consequently, the domain layer tends to highly coupled code with tangled concerns, which impacts error-proneness and maintenance of a system.

In conclusion, this architecture prefers transaction-oriented implementation of the domain layer with the Anemic Domain Model. It results in concerns scattered throughout a system and highly tangled together, which prevents their automated reuse. In addition, vertical repetitions make concerns reuse more challenging, which usually results in manual concerns repetition [A.3]. Unfortunately, none of mainstream technologies or standards consider business rules, UI widgets, or layouts as orthogonal concerns, and let developers tangle the concerns into source code manually instead, which significantly increases development costs and efforts [32, 43]. Consequently, the approach introduced in Chapter 4 addresses the separation of concerns in information systems.

3.1.3 Model-View Separation Principle

The other common organization of informations systems builds on the Model-View Separation principle [67]. This architecture assumes Rich Domain Model [40, 85] completely describes business domain and encapsulates its operations and related concerns. Then, there exist various views indirectly manipulating the model[1] and displaying it to users or exposing via APIs. Contemporary systems use many variants and implementations of this architecture; the Model-View-Controller [42], and the Model-View-Presenter [87] belong among the most common implementations[2]. Although there exist other architectures, assumption of the RDM makes this architecture a significant opposite of the layered architecture, which relies on the Anemic Domain Model [28, 40]. Therefore, this section elaborates qualities of this architecture and discusses its impact on the overall code organization and distribution of concerns within a system.

The RDM motivates to separate the UI concern and builds a strong and cohesive domain model maintaining logic and behavior. This allows execution and verification of business rules from all parts of a system, which eliminates repetition of the concerns and enables object-oriented decomposition techniques to keep design clean and simple. This architecture emphasizes object-oriented design and encapsulation of concerns to enable their reuse, which differs from the layered architecture. On the other hand, this approach suffers from the limited separation of concerns and significant repetition of business rules and possibly other concerns. Views usually use different technologies than a model, which prevents simple invocation and reuse of concerns. Then, the model must either implement and tangle technology-specific representations or expose the concerns for further transformation. Unfortunately, a complex domain model with encapsulated but tangled concerns significantly limits their automated inspection, subsequent vertical transformation, and reuse, which results in manual repetition of business rules and other cross-cutting concerns [A.10].

In conclusion, while the Model-View Separation principle evolved from the user interface, it is applicable on the large-scale architectural level as well as in distributed user interfaces. It is more costs-effective in long term projects compared to the layered architecture, which suffers from the issues of the ADM [85]. Although there exists a possible mapping of responsibilities between the Model-View Separation principle and the layered architecture[3], the reduced ability of the former architecture to inspect the concerns significantly limits their further reuse. In consequence, although the approach proposed in Chapter 4 is applicable to architectures based on the Model-View Separation principle, its efficiency and benefits are not significant due to fewer repetitions, and limited concerns inspection and automated transformation.

[1] Manipulation of the Model depends on an implementation of the separation principle. For example, in Model-View-Controller the Controller manipulates the Model and the View displays it.

[2] There exist various mutations implemented by contemporary web frameworks in many programming languages. For example, Nette for PHP (nette.org), Django for Python (djangoproject.com), Rails for Ruby (rubyonrails.org), and Play for Java/Scala (playframework.com).

[3] The Model relates to the domain layer and the View to the user interface templates in the presentation layer of the layered architecture [67].

3.2 Limitation of Conventional Development

This thesis is motivated by limited separation and reuse of cross-cutting concerns in conventional development of EISs. These systems implement business processes and maintain large volume of data to optimize a business domain. However, the domain, its policy, and requirements continuously evolve in time, which puts high demands on the maintenance process as these systems require continuous evolution [42, 43]. Since contemporary standards and best practices in software engineering suggest incremental iterative development [56, 70], the development process consists of high number of small iterations modifying the existing system, which is essentially equal to maintenance requests. Consequently, the system is permanently maintained [55], and thus it is essential to have optimal maintenance process. Therefore, this section highlights limitations in maintenance of a conventional EIS to be addressed later in this thesis.

Each maintenance cycle consists of several steps including the analysis of requirements, design of a solution, its implementation, testing, deployment, and evaluation [67]. The analysis addresses the requirements considering all kinds of concerns. It includes non-functional requirements and user experience, but also use case scenarios, access policy, and domain restrictions. During the analysis, the requirements are usually decomposed into independent concerns [76]. For example, redesign of UI widgets does not interfere with validation rules in a domain model on the requirements level, which indicates that the concerns are separable [A.3, 105] and can be managed by domain experts. For instance, user scenarios can be written by business experts, UI designed by UI designers, and the system architecture designed by system architects.

This separation of concerns at the requirements level arises from their mutual orthogonality [76]. Most concerns such as UI widgets and the application model evolve independently but some are related, e.g., the model and business rules [105]. However, consider the example in Figure 1.4. Each EIS represents a multidimensional space where each captured concern, including the application state, is one dimension. Unfortunately, the implementation space of source code is linear, which forces developers to linearize the multidimensional space into source code [83]. As a result, the concerns get tangled together (Problem 1), which produces complex source code with high amount of duplications (Problems 3 and 5). Furthermore, the complex code in a general-purpose language excludes the experts from the development process.

In consequence, the concerns are duplicated through the whole system, all layers, components, and multiple languages[4] as is discussed in Section 1.2. This limitation of the architecture, which is illustrated in Figure 3.1, is responsible for its progressive deterioration in time because of the difficulty to keep the design clean and all places synchronized, especially in constantly evolving systems. Although there exist tools,

[4] Each component may use different language to ease its maintenance. For example, data layer often uses SQL or some of its derivates and a user interface often uses HTML or similar language.

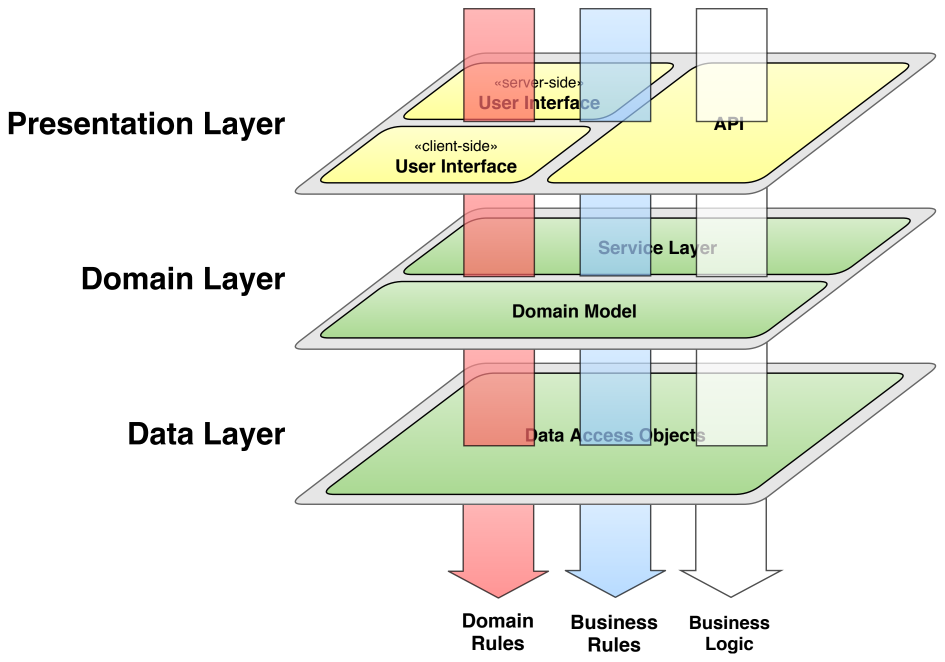


Figure 3.1: Cross-cutting concerns in the layered architecture

patterns, and architectures to ease the transition from the multidimensional space at the design level to the linear space of source code [42, 63, 98], conventional development often fails to use them and developers maintain the duplications manually.

3.3 Approaches to Separation of Concerns

Enterprise information systems are complex applications implementing business processes, maintaining large volume of data, communicating with other systems, and exposing data and processes to end users. Consequently, many concerns within these systems cross-cut together in multiple places. For example, the user interface presents data from a domain model and enables their modification considering current business rules. Data is presented via UI widgets and organized into UI layouts, and finally, each message within the UI is localized into the user's language. In conclusion, even a simple information system deals with significant amount of concerns. Although conventional approaches often fail to separate concerns, alternative approaches and techniques emphasize the concerns and focus on their representation and reuse.

The object-oriented programming (OOP) defines a set of principles to distribute logic among objects [67, 74], which naturally maps most real-world cases. There are plenty object-oriented (OO) best practices and techniques to handle growing structural and behavioral complexity [42, 45, 46, 88]. However, the challenge of separation of concerns arises from their natural tendency to cross-cut multiple objects, layers, components, or with each other, which makes them difficult to encapsulate in a single component using object-oriented programming [62, 104]. Moreover, not only the OOP suffers from concern tangling. Usually, a programming paradigm defines a dominant concern, e.g., classes in the OOP or functions in functional programming, and the other concerns become difficult to separate [83]. This results from a linear implementation space, while the concerns are usually orthogonal to each other and thus compose a multi-dimensional space [105]. In consequence, developers linearize the space as is illustrated in Figure 1.4, which results in a significant amount of repetitions, concern tangling, and error-prone maintenance as is discussed in Chapter 1. Fortunately, the following approaches provide mechanisms for separation, encapsulation, and automated integration of concerns in order to reduce development and maintenance efforts.

3.3.1 Model-driven Development

The model-driven development (MDD) maintains various models to capture, separate, and model system concerns and reduce manual repetitions [63, 99]. Each model describes a system from a different perspective and on a different level of abstraction. Then, models are semi-automatically evolved via forward transformation rules producing more specific models, which are manually refined and completed. In the end, source code is the most specific model of a system. Obviously, the MDD is driven by forward transformation, which is challenging. Once the system is deployed, it still requires further evolution. Unfortunately, lack of support of backward transformation makes propagation of modifications into more abstract models difficult, and thus regeneration of specific models overwrites the manual modifications.

The MDD often uses UML [96] as a graphical modeling language to reduce mental barrier and increase efficiency of a modelling process. However, case studies show that use of UML is no faster than simple coding and the speed of development with UML varies from -40 % to +15 % comparing to development without UML. For example, a case study conducted in [31] shows insignificant 14 % decrease in performance in exchange for 54 % greater functional correctness. Since standard UML is not descriptive enough, the MDD often utilizes other languages such as the object-constraint language (OCL) [81, 114] to represent business constraints. Alternative implementations of the MDD separate the concerns via integration of the aspect-oriented programming [65] and utilization of CASE Tools [77, 106]. Such approaches assume maintenance of concerns on various levels of abstraction with automated forward transformation, which is supposed to make maintenance easier and available to domain experts.

In conclusion, while the MDD aims on separation of concerns, it suffers from significant limitations of semi-automated forward transformation, and UML expressiveness. Moreover, it usually follows object-oriented principles, which are unable to efficiently separate and reuse cross-cutting concerns [62], and adopts their limitations [78]. Therefore, the models still tangle concerns together, which results in manual repetitions and difficult maintenance of a system.

3.3.2 Aspect-oriented Programming

The object-oriented programming proved to be unable to address concerns cross-cutting via multiple methods, classes, objects, layers, or components [105]. Consequently, the aspect-oriented programming (AOP) was designed to complement the OOP to directly address the separation of cross-cutting concerns and to reduce software complexity [62]. It provides an alternative mechanism to modularize a system and isolate secondary and supporting concerns from the main business logic.

While the OOP uses classes to structure and encapsulate logic, the AOP encapsulates the cross-cutting concerns in aspects. These are independently described concerns in any convenient, usually domain-specific, language. The AOP identifies various joinpoints, which are places in source code (static joinpoints) and an invocation flow (dynamic joinpoints) available for concerns injection [112]. Then, there is mapping (the pointcuts) selecting subset of joinpoints for concern injection. Finally, the concerns are automatically weaved into the joinpoints and distributed throughout a system [62, 66].

There are two major approaches to aspect weaving. First, compile-time or load-time weaving performs aspect integration during system compilation or initialization, since the weaving process can be quite expensive and this approach executes it outside the running system. On the other hand, the aspect weaver may consider only static information, i.e., the result is contextless. For example, the AspectJ framework[5] accepts aspects written in Java [66], it takes Java bytecode, and enriches it with compiled aspects. The load-time weaving is applied by the Spring framework[6] as simplification of a development process. It postpones compile-time weaving of AspectJ and provides the modified classloader performing load-time weaving, instead. The other approach considers dynamic joinpoints and weaves aspects at runtime [83, 104]. The weaver accepts contextual information and thus the aspects cannot be weaved outside the context [A.3]. Although this is expensive[7], the resulting context-aware code delivers desired behavior and enables more complex combinations of concerns, while source code keeps concerns encapsulated, separated, and without any duplication.

In conclusion, while the AOP is a complement to the OOP, the provided benefits significantly impact development and maintenance efficiency. Ability to separate and then weave concerns reduces the amount and complexity of source code as well as number of repetitions and thereby positively impacts source code quality [32, 43, 105]. Unfortunately, the requirement of an aspect weaver and convenient languages for concerns description introduce a major initial barrier. However, the approach introduced in the next chapter significantly utilizes the AOP in order to address separation of concerns and problems identified in Chapter 1.

[5] Available at eclipse.org/aspectj.
[6] Available at spring.io.
[7] Web applications enable concerns distribution and caching, which moves aspect weaving to client devices and significantly reduces server load [A.12].

3.3.3 Domain-specific Languages

The aspect-oriented programming allows aspects to be described in any efficient language if the weaver accepts it [62]. These kinds of languages are formalized as domain-specific languages (DSLs) [41], which introduce wide range of advantages. Most significant benefits are efficient domain description as the language is tailored exactly for that domain, and possibility to delegate responsibilities to domain experts as the language might be simple and easily understandable. This technique is intensively utilized, although it is not often referenced as DSLs. For example, HTML, SQL, and plenty configuration files of all sorts of software belong among these languages.

Although there are valuable benefits, DSLs face couple of challenges [41, 72]. Complex design and implementation belong among the most significant challenges as the process is difficult. It involves many issues including grammar specification and interpreter/compiler implementation. Furthermore, the more languages a user must know the more difficult and inefficient it gets. Next, although easily describable domains are great, when a new user starts using the system with multiple DSLs, there is very steep learning curve significantly reducing those benefits. Nevertheless, DLSs proved themselves to be an excellent tool for domain modeling and code generation and are utilized in various approaches discussed later in this chapter [A.3, 58].

3.3.4 Domain-specific Modeling

The domain-specific modeling (DSM) is another approach to separation of concerns. It assumes development of multiple systems within a same business domain, and then it optimizes a development process via utilization of a DSL for domain description. The language is tailored iteratively to deliver best possible performance and the model is used for code generation, which brings significant reduction of development efforts [58]. The domain-specific modeling, separation of concerns, and automated code generation increases overall efficiency and reduces the amount of manually maintained repetitions.

Comparing to the model-driven development, there exist two major differences. First, while the MDD uses the UML with the OCL for constraints, the domain-specific modeling deploys domain-specific languages to reach higher efficiency and ease description of domains. For example, it involves templates, special mark-up derivatives, and even a graphical language to describe the domain. Another difference lies in the code generation. While the MDD uses static forward transformation of the models, the domain-specific modeling produces the code even at runtime considering the current runtime context. This brings more dynamics into models, while it also removes the limitation of forward transformation. With a powerful but complex code generator and suitable DSLs, it is possible to avoid or significantly reduce manual code duplication and achieve simple and easily modifiable source code [58].

In conclusion, while the domain-specific modeling provides an efficient mechanism for domain modeling and code generation, it assumes production of multiple systems within the same domain to meet claimed efficiency and utilize initial efforts. Unfortunately, the first applications as well as the underlying domain-specific mechanism are still manually developed in a general-purpose language and the domain-specific modeling is derived later using gathered experience.

3.3.5 Generative Programming

Generative programming (GP) directly addresses the separation of concerns [25]. It decomposes a system into components encapsulating separated concerns, and then the parametrized build dictates the rules to construct the system from those components, i.e., to integrate the concerns back together. However, GP does not address a runtime context and constructs the system at compile time instead. Then, the build must produce all combinations, which may occur at runtime. In consequence, although GP separates the concerns, transforms templates into the resulting system, and reduces development and maintenance efforts, the number of produced combinations grows exponentially with the number of concerns and components, which makes it inefficient for large information systems with a significant amount of context-aware executions.

3.3.6 Concern-driven Development

Concern-driven development (CDD) [2] is another approach addressing the separation of the concerns and their automated reuse. This approach extends the model-driven engineering [98] and describes each concern in a convenient concern-specific model in a most generic project-agnostic form, and thus makes it reusable across various projects. Having the concerns described in the models, the approach introduces the Concerns Library, which is a toolbox of available concerns [2]. Then, implementation of a system is only a combination of existing variances of concerns, and concern-specific work can be left to domain-experts maintaining the model.

In order to separate the concerns into independent models, the approach utilizes the aspect-oriented programming. It extends the aspects and uses aspect-oriented user requirements and reusable aspect models, which implement concerns [68]. While the idea of leaving concerns maintenance to domain experts is attractive, in fact, it is very difficult to describe the concerns in such a generic form to be able to reuse them. Usually, this generality comes with high complexity, which increases error-proneness and makes maintenance tedious and challenging.

In conclusion, this approach separates the concerns into independent concern-specific models, and then applies transformation rules to automatically transform the models across various levels of abstraction to avoid manual duplication and enable automated concerns reuse. Moreover, encapsulation of concerns in a toolbox reduces complexity of system development and allows software engineers to work on a higher level of abstraction, while it leaves concerns maintenance to domain experts, instead. Unfortunately, the complexity of models and the approach itself significantly increases its initial overhead and reduces overall efficiency [2].

3.3.7 Summary

Although there exist approaches addressing various aspects of separation of concerns, it is still a challenging task. Unfortunately, there is no approach tailored for the domain of enterprise information systems, which operates with a specific architecture and only with some types of requirements. Therefore, contemporary information systems either combine these approaches to the separation of concerns or use a naive implementation, which results in significant code duplication, error-proneness, and increased efforts.

Despite the absence of a specific approach, some combinations of approaches deliver significant improvement of development process. For instance, the OOP and the AOP are implemented by the Spring framework and the OOP with the MDD are implemented by various CASE and MDD tools [98]. Unfortunately, none of these recognizes the importance of business rules within a system. Therefore, the following chapter proposes the aspect-driven development approach to design of an enterprise information system. The approach utilizes existing approaches and programming techniques to address the separation of concerns in enterprise information systems and Problems 1 to 5 stated in Chapter 1. While this section summarizes the existing approaches, their background, benefits, and limitations, their influence on this work and the adopted principles are briefly suggested in Section 4.1. To complete the elaboration of the approaches, consider their impact on the terminology in this thesis:

- The concern-driven development considers the concerns to be an important part of the software. It operates with the term *concerns*, which is adopted by this work. It considers the concerns and, comparing to the conventional understanding, increases their importance in the scope of information systems.
- The model-driven development emphasizes the importance of the models and uses several levels of abstraction. Then, it uses *forward transformation* to produce more specific models, which is also adopted in this thesis in order to reuse concerns in multiple components and technologies.
- The domain-specific modeling contributes to the background of the approach introduced in the next chapter. It utilizes the DSLs to reduce the mental barrier, involve domain experts, and increase overall efficiency. Similarly, the generative programming also operates with the identical objective, considers the concerns, and generates the code, but it uses a different mechanism, thus there is no significant intersection with this thesis.

- Finally, the significant contribution is made by the aspect-oriented programming. This approach is the underlying mechanism utilized in the next chapter. It defines a major part of the used terminology. The aspects in the AOP are the concerns within this thesis and thus the terminology around the aspects is fully adopted. The *aspect weavers* are the components responsible for the concerns composition, and the *joinpoints*, *pointcuts*, and *advices* are discussed later. Moreover, there is defined formal mapping of the proposed approach into the AOP.

In conclusion, this thesis is influenced by several approaches. The concerns, their separation, and runtime composition are the key concepts of the approach proposed in the next chapter. Unfortunately, the term concern-driven development already exists [2]. Concerns are basically the aspects, therefore, naming the approach aspect-oriented development would fit the idea but the aspect-oriented software development also already exists [37]. Finally, the *aspect-driven development*[8] still fits the purpose and is not in direct collision with the existing terminology.

3.4 Separation of Concerns in a Conventional EIS

While this thesis addresses the separation of concerns in enterprise information systems to avoid concerns tangling via their separation and automated reuse, even existing conventional information systems partially deal with these concerns. The previous section discusses general approaches to the separation of concerns, this section elaborates existing approaches and techniques on a lower level of abstraction. It considers common components of a conventional system, since the approaches and frameworks usually focus on a single component instead of a whole system and its design.

3.4.1 Concerns in the Domain Layer

Although the domain layer is in the middle of the layered architecture, it implements domain model, which drives an entire information system. It is defined by business domain of a system and interacts with business processes. This layer implements the processes and as such, it is the essential layer in a system. Other layers facilitate persistence or provide access and manipulate data. Since this layer implements business logic with all constraints, it is elaborated first as it impacts the rest of a system.

Business logic is difficult to describe and maintain [15]. Its efficient isolation and subsequent reuse using pure object-oriented techniques is challenging even inside the domain layer, since it belongs among the cross-cutting concerns [24, 73]. The issue of business logic maintenance in the layered architecture is often connected to the

[8] The papers publishing the research in this thesis reference the aspect-driven development as the aspect-oriented design approach (AODA) and the aspect-driven design approach (ADDA). The name evolved with the underlying idea and its connections to other existing approaches.

Listing 3.1: Input validation in the Transaction Script design pattern

```java
public Product createProduct(Product product) throws Invalid {
  // verify name of the product
  if (product.name == null || product.name.length > 200) throw new
      Invalid();
  // verify price of the product
  if (product.price <= 0)  throw new Invalid();
  // verify ordering weight of the product
  if (product.weight < 0)  throw new Invalid();
  // instance is valid, save and return the product
  return em.persist(issue);
}
```

Transaction Script design pattern [42], discussed above in Section 3.1. This pattern encapsulates business logic together with business rules in use case-oriented separated transactions representing business operations. Unfortunately, this tends to code tangling and constraints repetition as is shown in Listing 3.1. Furthermore, it does not allow any constraints reuse. The impact of the Transaction Script on business logic tangling can be partially reduced by object-oriented principles. When there are two transactions sharing the same logic, it can be refactored out into a separate object [43]. Such an approach allows partial reuse of code, although transactions usually still remain hard to read and maintain. Capturing model-related validation rules independently as a separated concern, and having other rules in validator objects eases reuse of business rules within the layer [15], reduces code repetition and error-proneness.

Alternatively, Java standard *JSR 303: Bean Validation* [8] utilizes declarative programming and meta-programming techniques to separate and reuse validation constraints. This standard introduces declarative constraints attached to model fields, which is illustrated in Listing 3.2. These constraints are verifiable by constraints validators[9]. Standard JSR 250 [75] introduces a new annotation `@RolesAllowed` annotating business operations. This annotation declares accepted security roles to restrict access to the operation and ease security implementation in the domain layer[10].

Unfortunately, although these declarative rules seem reusable, there are major limitations. They are designed to annotate the model, i.e., capture there all invariant constraints, which are always verified and conditions must be satisfied. In fact, preconditions (input validation rules) of business operations vary. They are determined by the position of the operation in the modeled domain and impacted by current execution context, i.e., the application state and user's context, as is discussed earlier in

[9] For example, Hibernate Validator implements JSR 303 standard and extends it by its own annotations allowing constraining fields by additional common restrictions. (hibernate.org/validator/)

[10] In the domain layer, Java EE uses Proxy and Chain of Responsibility design patterns to capture invocations of methods and check for methods annotations to trigger attached behavior such as input validation or access restriction [28, 45].

Listing 3.2: Annotated domain model with JSR 303: Bean Validation

```java
public class Product {
  public Long id;
  @NotBlank @Length( min = 10, max = 100 ) public String name;
  @NotNull @Positive public Double price;
  @NotNull @PositiveOrZero public Double weight;
  @NotNull @PositiveOrZero public Integer inStock;
  public LocalDateTime validUntil;
  // constructor, getters and setters
}
```

this work. In conclusion, the extension is unable to express any of these rules, which falls back to code tangling.

In conclusion, although there exist tools and approaches to efficiently encapsulate and reuse business rules, their implementations suffer from significant limitations. In consequence, although they claim more efficient development, it is either very complex or fails to address most of business rules, which usually leads developers to manual repetition and tangling of the rules into source code.

3.4.2 Concerns in the Data Layer

The data layer in the layered architecture facilitates persistence and provides abstraction over remote sources. Implementation of such functionality is usually straightforward and often partially automated, but often lacks some features as it does not address the separation of concerns. First, although it should protect a persistent storage and verify integrity constraints, they often absent or are manually repeated. Second, although each operation has its context and there exist a set of preconditions and postconditions, these constraints are usually manually repeated in the calls instead of being automatically reused from the context of the operation. Consequently, this section elaborates separation and reuse of these concerns in the data layer in order to reduce manual code repetition and ease development and maintenance of a system.

Leading platforms for development of EISs such as Microsoft .NET and Java Enterprise Edition use Object Relational Mapping (ORM) frameworks [3, 27] to map a domain model into a relational database. This mapping simplifies use of a persistent storage, e.g., a database by making an abstraction level above it. For example, Hibernate [6] is an evolved ORM framework extending the Java Persistence API standard [27] by additional restriction declaration. It utilizes JSR 303: Bean validation and transforms these constraints such as `@Email` or `@NotBlank` into database integrity constraints, when it produces a database schema. Moreover, the constraints are usually easily verifiable at the entrance of the data layer as it is usually implemented in the same language as the domain layer.

On the other hand, there is no existing framework reusing business rules and constructing queries from a business context. Although Hibernate transforms annotations of the Java Bean Validation [8], which supports a small part of business rules, into integrity constraints, it does not support business contexts themselves, nor contextual business rules. Instead of introducing a business context on a higher level of abstraction and focusing on its system-wide transformation and reuse, existing tools and frameworks remain in the data layer and focus on design and development of domain-specific languages to ease manual definition of queries [27, 103][11]. Although this activity reduces efforts and error-proneness of development, manual repetition of concerns and implementation of a DSL in Java or a similar language still maintain a significant barrier for domain experts.

In conclusion, while there exist tools and generators accepting a domain model[12] or a protocol of a remote service[13] and producing an implementation of the data layer, they often fail to reuse business rules. Although Hibernate Validator is able to verify JSR 303-based constraints and transform them into integrity constraints, it is unable to construct a query from a business context. Other existing tools also focus more on improving efficiency of development of the data layer instead of improving efficiency of an entire information system.

3.4.3 Concerns in the User Interface

The presentation layer is the most complex layer of an enterprise information system with the layered architecture. It exposes data and business processes to end users and other systems, and thus provides various graphical and programmatic APIs. The APIs reflect the structure of the domain model and business rules of business processes to facilitate up-to-date interface, which validates input data, and provides a user-friendly or a system-friendly error in case of violation of the rules. The UI considers additional concerns such as layouts, widgets, and localization, which cross-cut throughout the whole component [17]. Linearization of this multidimensional space into source code significantly degrades design and introduces information repetition [105]. As such, this section elaborates cross-cutting concerns and their separation in the UI, which is the most complex component of the layer; it is often distributed among a server and one or multiple clients, and implements multiple cross-cutting concerns tangled together. The UI of enterprise information systems is a great challenge, since it consists mostly of input forms to present data and enable users to manipulate them. In consequence, this section discusses the separation of concerns in the UI with the focus on input forms

[11] In addition, there exist many frameworks for various languages providing a query DSL. For example Squeryl (squeryl.org) and Quill (getquill.io) for Scala, and Querydsl (querydsl.com) for Java.

[12] For example, Data Access Object Code Generator for Java (titaniclinux.net/daogen).

[13] For example, Apache CXF (cxf.apache.org) produces models and stubs of Data Access Objects for remote services described in WSDL or WADL schema.

as they constitute the major and most complex part of the UI. The other components of the presentation layer are usually much simpler, thus not considered here.

Conventional development approaches such as Java EE [28] lack the ability to separate and reuse concerns in the UI. Java standard *JSR 303: Bean Validation* [8] utilizing declarative meta-programming enables limited reuse of business rules. The JBoss RichFaces[14] extending JavaServer Faces [13] inspects a domain model and gathers declared constraints. Then, it transforms them into the client-side of the UI, which delivers better user experience without any additional efforts or maintenance issues. Combination with the Hibernate Validator also facilitates server-side validation and fully benefits from reusable model constraints. Therefore, despite limited expressiveness, and absence of support of business operations, and context-dependent constraints, it still significantly reduces business rules repetition [A.14]. Unfortunately, conventional approaches do not address other concerns, and manually duplicate code and tangle concerns, instead.

Metawidget[15] is an alternative approach to input forms generation in the UI. It identifies concerns such as a model structure, validation rules, UI widgets, and localization [59], and assumes that all these concerns can be described in the domain model. The approach relies on model inspection and conducts software mining [61]. The UI is constructed from the mined meta-model by pluggable processors such as widget and layout builders [60]. Although the approach focuses on automated UI generation and self-maintainable UIs, it does not strictly separate the concerns. It utilizes concern-specific builders and inspectors but it hits significant limitations in their composition. This approach does not consider runtime context such as user's roles and a geographical location to perform additional UI transformation, e.g., to show an additional input box in an address form to select the state within the United States.

Alternative Rich Entity Aspect/Audit Design[16] (READ) approach to input forms generation combines the model-driven development, the aspect-oriented programming, and the generative programming [A.3, 16]. It directly addresses the separation of concerns in the UI, and recognizes concerns such as a model structure, UI layouts, UI widgets, and localization, but does not address business rules nor input validation. The approach separates the concerns as independent aspects [62], describes them in an efficient DSL [41, 72], and weaves them together at runtime considering the runtime context. Similarly to the Metawidget, the READ inspects an instance of a model and constructs a meta-model enriched with contextual information. Then, the aspect weaver accepts the concerns as a multidimensional space and a mapping (pointcuts) considering runtime parameters. Finally, the resulting source code fragment is injected into the UI. Regarding the efficiency, conducted studies show significant source code

[14] Available online at richfaces.jboss.org.
[15] Available online at metawidget.org.
[16] Its AspectFaces implementation is for Java EE applications is available online at aspectfaces.com.

reduction up to 32 % [A.3]. Unfortunately, limited support of business rules leads developers to JSR 303 with all its benefits and limitations.

Contemporary EISs stress development of native mobile applications backed by an API. While there are several target platforms, application logic and views are usually same regardless the platform. Since the platforms differ significantly, it is difficult to reuse configuration and code. The READ approach was extended for distributed concerns delivery, which enables modern self-standing UIs with reuse of server-side concerns description [18]. The server provides a meta-model of an instance including the concerns to be rendered and the weaving is performed at the client's side. Conducted benchmarks show a significant performance improvement with distributed UIs [A.12]. It reduces server load and allows use of multiple native clients without any concerns restatement or code duplication. For example, a single instance of a domain model is transformed into a contextual meta-model and rendered as an input form in native clients, which are implemented in JavaScript for web browsers, in Swift for iOS, and in Java for Android. While the case studies show promising results, efficient development, significant code reduction, and support for distributed UIs, limited support of business rules still results in their manual repetition in code.

The model-driven approach to development of distributed UIs maintains models on four levels of abstraction to reuse concerns and produce multiple UIs for different platforms and contexts of use [108]. It describes those models in the UsiXML DSL and supports both forward and reverse model transformation to eliminate previously stated limitations of the MDD. Unfortunately, this technique prevents reuse of concerns outside of the UI. Similar model-driven technique is proposed in [22]. The framework expects a system captured in UML diagrams, which are transformed into source code for multiple platforms. However, this technique suffers from inability to efficiently express runtime context and deliver context-aware UI [5].

Although there exist approaches addressing generation of input forms in the UI via the separation of concerns, they significantly differ. While both the READ and the Metawidget perform runtime inspection, the Metawidget has pluggable builders in a native programming language, while the READ performs full concern decomposition allowing their description in DSLs, which is more efficient. Contrary to the Metawidget, the READ lacks the support of validation constraints and business rules. Alternative approaches to development of distributed UIs are usually model-driven, but do not address the problems outlined in Chapter 1. They either do not address business rules as a concern, or the generation process is not context-aware. In consequence, although the approach introduced in Chapter 4 provides a system-wide concerns reuse, it is inspired by the READ, which already addresses multiple identified problems.

3.4.4 Concerns in the Service-oriented Architecture

Complexity and wide use of enterprise information systems emphasize their robustness and scalability, thus the scope of a system often exceeds a single application. As a result, application logic is decomposed into small, standalone, and loosely coupled[17] *services*, each encapsulating a part of the business domain. These services are then composed together into *composite services* to deliver more complex functionality [84]. In total, they create a large distributed system in the service-oriented architecture (SOA)[18] [86]. Since the SOA suggests a system infrastructure following the structure of the real business [86], then considering the running example, the departments responsible for the store (products and stock) and customer service (orders) are components in the system [67, 84]. The internal design of these components may vary based on their implementation, but the properly implemented SOA emphasizes scalability and testability because it significantly reduces complexity of services comparing to monolith applications [30]. On the other hand, it makes information reuse challenging, which usually leads to its manual repetition instead, which makes the SOA prone to inconsistencies and expensive maintenance. Therefore, this section elaborates common approaches to separation and reuse of concerns in the SOA, since Chapter 6 implements the approach from Chapter 4 into a distributed environment.

Definition 3.1 *A service is a reusable, cohesive, managed, deployable, and independent process interacting via messages [30, 84].*

Definition 3.2 *A composite service accesses and combines information and functions from existing service providers [84].*

Definition 3.3 *The service-oriented architecture is a set of design principles organizing software components (services) around business capabilities and connecting them through standard interfaces and messaging protocols. Each component is self-contained, black box for consumers, and exposes only its interface [44, 84].*

Decomposition of a system into small units delivers simpler design, evolution, and maintenance of units and the system itself [116]. Communication through a neutral environment such as HTTP protocol makes services independent of a particular technology, which enables development of each service in a different programming language

[17] The services interact through a REST-like protocol or a messaging bus and use catalogs and service locators for discovery [33, 36, 44, 91].

[18] Contemporary systems often follow more evolved specializations such as Microservices [30, 44, 101] due to poor implementation of the original SOA and many failed projects in past. However, this work considers the SOA itself and other architectures are left for future work.

and a technological stack. The natural decomposition by a business structure clearly defines responsibilities of components, which supports agile programming [67], multi-team development, and rapid delivery. Nevertheless, challenges to face still remain and some of them are addressed in Chapter 6.

Loosely coupled services introduce a challenge of discovery of their dependencies. Conventional systems are either orchestrated in a network with a director validating and forwarding messages, or the services know their dependencies and look them up themselves (choreography) [30]. Recent approaches to service composition utilize Artificial Intelligence due to increasing number of existing services and their complexity [91]. In addition, since there exist many implementations of service description, discovery[19], and meta-data extraction techniques, it is very difficult to gather and reuse this information. In consequence, although composite services are aware of existence of their dependencies and consider a subset of their concerns, neither the organization of a network nor the discovery process address reuse of concerns.

While decomposition brings clear design, a composite service must reuse information from its dependencies. For example, *Order* composite service needs to know the model structure and the business rules of the underlying services *User* and *Product* to be able to validate incoming orders or even additionally transform and expose the rules to web API, e.g., for client-side validation necessary for a user-friendly UI. However, distributed environment and possibly different technologies basically prevent simple sharing. The model description can be exposed through API schema[20], but reuse of business rules is very difficult even in monolithic applications [15, A.8]. There exist model-driven approaches for the SOA [90], but lack the support of business rules [57]. Another approach proposes an extension of the Business Process Execution Language (BPEL) [4] to separate business rules from services, since it considers them to be a subject of change [93]. It emphasizes their separation and automated reuse, but the BPEL is designed for a centric orchestration, while recent research and best practices suggest decentralization through a choreography [30]. Finally, a framework for construction of composite services introduced in [35] identifies a business context and reuses business rules from dependencies; although it is not generic and requires higher amount of manual work, since each dependency must be manually described in order to be reused. In conclusion, existing work on reuse of business rules in the SOA is limited and requires significant amount of manual work.

In conclusion, while the service-oriented architecture and its evolved variants decompose a system, enable parallel evolution of its components, and support multi-team development, they introduce additional challenges. Considering the scope of this the-

[19] For example, service are discovered through a service registry such as Universal Description, Discovery, and Integration (UDDI) catalog, their addresses are hard-coded into source code, or the configuration is provided by a central component [71].

[20] There exists the SOAP with WSDL [19] or RESTful services optionally with WADL [50].

sis, especially technological diversity and encapsulation of services significantly impact separation of concerns and their automated reuse, which results in Problems 1 to 5 identified in Chapter 1. The approach introduced in Chapter 4 for monolithic information system is adjusted in Chapter 6 for distributed environments, and addresses the outlined problems in order to reduce manual repetitions and maintenance efforts.

3.5 Representation of Business Rules

Since enterprise information systems implement business processes of a business domain, they deal with significant amount of business rules. These rules constrain the processes, their tasks, and put constraints on a domain model. Unfortunately, since EISs are tailored to maintain and manipulate data, the rules cross-cut throughout the whole system. Therefore, separation, transformation, and automated reuse of business rules is essential for efficient implementation and subsequent maintenance. However, representation of business rules is a challenge itself and considering the scope of information systems, various contexts and technologies, it becomes even more challenging. Therefore, the approach introduced in Chapter 4 discusses representation and reuse of business rules in detail in Chapter 5, and defines complex requirements on a language for their representation. Consequently, this section briefly elaborates existing business rules representations in order to evaluate their suitability for inspection, automated transformation, and reuse to address the problems and objectives set in Chapter 1.

3.5.1 General-purpose Languages

Contemporary EISs often do not emphasize business rules and tangle them into source code in a general-purpose language (GPL) together with business logic in the domain layer. While GPLs efficiently describe behavior of an entire system [110] and are able to implement any business rule, there exist significant limitations reducing their efficiency for business domain description [A.11, A.7].

Complexity and maintenance of business logic represented in GPLs is discussed in [15]. Together with analysis of layered EISs, the authors propose decoupling of common aspects such as exception handling from the rest of the system to have logic in a single place and to reduce maintenance effort. The proposed solution proceeds from design patterns [45, 88], but suffers from some code duplication.

Considering suitability of GPLs for the novel approach and stated objectives, there exist serious limitations preventing GPLs to be an efficient representation of business rules despite their nearly unlimited expressiveness. First, GPLs tend to be complex, difficult to learn, and require analytical thinking and advanced expertise. This complexity prevents domain experts from maintaining the domain because they are experts in their domain, not in programming languages. Second, GPLs used for implementation of EISs are usually imperative and intended to be executed but the rules are declarative as a rule may have different implementations in various contexts, since they are transformed into various layers of a system. Furthermore, GPLs are designed to be compiled/interpreted but not inspected and transformed, which is essential for automated distribution of the rules throughout a system.

3.5.2 Meta-programming

Unfortunately, object-oriented programming languages fail in implementation of cross-cutting concerns and distribution of functionality repeated in a system [62]. In order to deal with this limitation and avoid manual code repetition, the languages provide alternative mechanisms to create hooks and inject functionality such as meta-instructions. For example, Java provides annotations [12] and interceptors.

Java EE comes with *JSR 303: Bean Validation* [8] and *JSR 380: Bean Validation 2.0* [79] standards introducing annotations to put constraints on a domain model as is illustrated in Listing 3.2 and elaborated in depth in Section 3.4.1. It deploys declarative programming and meta-programming to separate and then reuse validation constraints in a system [34]. Unfortunately, although the standard aims on constraints reuse, there are no production-ready implementations for other layers of a system.

In conclusion, meta-instructions such as Java annotations are checked by a compiler, and inspect to extract business rules from source code, Unfortunately, they are unable to represent complex conditions and whole spectre of business rules due to reduced expressiveness, which in combination with difficult extendibility and local scope makes them unsuitable for representation of global rules applicable throughout all layers. Finally, presence of annotations in source code eliminates domain experts from development and requires recompilation of a system for changes to take effect.

3.5.3 Expression Languages

Need for expressing conditions including business rules in various parts of a system resulted in a proposal of *Simplest Possible Expression Language*, which is part of *JavaServer Pages (JSP)*. This scripting language provides a subset of instructions of a GPL and brings dynamic evaluation of expressions at runtime. This language evolved through an expression language (EL) for JSP 2.0 and an EL for JSF 1.1 into standardized *Unified Expression Language*, which is a part of JSP 2.1 [23].

There exist many implementations of the language for many platforms, which meets the need for transformation of business rules into various platforms and technologies. Unfortunately, the language is not type-safe, which results in error-prone development. Also its generality and complexity prevent involvement of domain experts into development, and domain-specific syntax sugar does not exist , thus use of such a language would be cumbersome. Although, the EL is able to define business rules and language parsers can be used to enable their transformation [10, 23]. Unfortunately, it is unable to describe a business context of an operation, i.e., a set of preconditions and postconditions. The complexity of such a context is beyond the expressiveness of the language. Finally, the EL is often combined with meta-programming, especially with Java annotations. Together, they attach additional reusable context-aware expressions to various components in source code, in order to reduce impact of limited support of cross-cutting concerns. In conclusion, although the EL provides some beneficial aspects, it does not fit to the objectives set in this thesis.

3.5.4 Domain-specific Languages

While the previous languages and techniques suffer in involvement of domain experts into development, the domain-specific languages (DSLs) overcome this limitation of GPLs and are tailored for a particular domain [54]. This reduced set of instructions focused on a single domain eases development and maintenance, while it preserves its own comprehensibility, reusability, and expressiveness [41]. Although growing number of languages within a system increases both mental and technical complexity. In addition, tailoring a language is a significant challenge itself, since the language must be properly designed and several tools such as parsers, compilers/interpreters and others must be implemented [72]. These are not trivial issues; however, once they are resolved, the language with the tools can be reused among many projects.

Simplicity and efficiency of DSLs and extraction of domain description out of application source code helps overcome both mental and technical barriers and let domain experts involve into development to maintain a domain of their expertise [41]. Having a DSL designed for a particular purpose in a platform-independent representation allows its inspection, further transformation, and context-specific evaluation, which are essential requirements for addressing the problems outlined in Chapter 1. On the other hand, DSLs tend to silently grow and adopt new concepts to cover more edge cases, which might significantly impact development in later phases. However, a case study conducted in [A.14] concludes that DSLs are an efficient choice for business rules representation, which is strongly confirmed by many rule-based systems [52] working with hundreds of business contexts described in a DSL. Application of domain-specific modeling, i.e., use of DSLs and specific generators, can increase the overall productivity by 500-1000 % [58]. Since there exist several major languages for business rules representation, the following text discusses their suitability for this thesis.

JBoss Drools[21] and similar languages for rule-based systems use business rule-oriented conceptual modeling, which works with Event-Condition-Action rules [11, 64] and utilizes all advantages of DLSs. Although these languages are comprehensive and self-documenting, they still tangle business rules throughout multiple actions, as they do not consider rules separation, encapsulation, and reuse. The rule-based systems optimize rules evaluation in expense of inspection, which makes them difficult to transform and reuse [38]. Finally, although these systems actually use a business context[22], they do not recognize it as a concept in terms of this thesis. Moreover, the comparison of rules represented in rule-based systems and rules within EISs shows that the rules in EISs are a subset of rules in the rule-based systems. In conclusion, although the languages for rule-based systems are efficient, they are too complex for EISs and miss some essential qualities arising from the definition of a business context.

Model-driven development (MDD) [63] uses domain-specific languages to describe models of various domains on different levels of abstraction. The authors in [77, 106] propose focusing on business rules and maintain them on various levels of abstraction to be maintainable by both developers and domain experts. They suggest maintaining the rules using CASE Tools and then referencing them from the code to allow their automated transformation. The authors in [55] recommend the *Don't Repeat Yourself* principle suggesting to capture every piece of information in code just once. An extension of the MDD applies Object Constraint Language (OCL) [29] into UML modeling and discusses its use within the MDD [81]. Although this approach overcomes several limitations of the UML and it is often used in research papers for its formalism, it results in complex UML models and thereby increases the barrier for domain experts.

Business process modeling uses Business Process Execution Language (BPEL) [4] and its graphic notation Business Process Modeling Notation (BPMN) [20]. It efficiently models the processes, possibly in a graphical language, however, it focuses on a higher level of abstraction. It orchestrates web services, components, and shared messages, and defines communication protocols, instead of modeling business operations and model constraints. An alternative modeling language for rule-based systems is the Simple Rule Markup Language (SRML) [107], which is a standard suffering from similar limitations as JBoss Drools. The ontology-based Semantic Web Rules [53] efficiently model business rules, but at the expense of required ontology definition, and complex transformation, which significantly increases all initial, intellectual, and implementation efforts. Furthermore, language complexity makes involvement of domain experts and documentation derivation more difficult. Nevertheless, these languages seem to deliver excellent business description when combined with BPMN [118].

[21] Documentation is available online at drools.org, a reference guide is in [97].
[22] A rule in rule-based systems follows the Event-Condition-Action pattern, where the Event, its attributes, and the Condition is basically a business context in terms of this thesis.

Finally, loosen syntax of some GPLs such as Scala, Ruby, and Groovy is convenient for design of a DSL [47]. A resulting language is comprehensive and easy to use, while remains type safe, if the original language is. Power of a GPL enables easy implementation of various aspects of a language, while invocation of a DSL constructs a model, which avoids the inspection. On the other hand, use of a GPL is platform-specific and might be difficult to use with systems on other platforms. Need for compilation/interpretation also increases a mental barrier for domain experts.

In conclusion, it shows that a DSL is an efficient approach for business context description. It fits the needs of separation of concerns in return for high initial investment into language design and implementation of related tools. Unfortunately, none of the existing languages addresses the specific requirements of this thesis, thus it is necessary to design and develop a new DSL, as is discussed in detail in Chapter 5.

3.5.5 Summary

There exist multiple kinds of business rules representation, which differ in qualities, intended use, and benefits, but also suffer from various limitations. While general-purpose languages are powerful, they are difficult for domain experts and their transformation is challenging. Meta-programming emphasizes declarative approach, which corresponds to intentions of business rules, but its use is cumbersome for domain experts and it shares various limitations with GPLs in addition to reduced expressiveness. While combination of an expression language and meta-programming delivers promising performance and efficiency, it is not statically checked. In addition, it is located in source code, and the expression language is based on a GPL, which are aspects preventing domain experts from maintenance of the concern. However, domain-specific languages seem to be efficient in business rules management. Since these languages fit a specific domain, they are friendly to domain experts, who may maintain it instead of developers. Unfortunately, designing a new language is challenging and it requires implementation of various supporting tools such as compilers and editors, which is a significant initial barrier.

In conclusion, since the approach proposed in this thesis considers business rules as a major concern of a system, evaluation of existing languages for business rules representation and choice of a language is essential. Efficiency of the language significantly impacts maintenance of the concern and subsequently overall development and maintenance efforts. While this thesis aims on design of a development approach for enterprise information systems, the challenge of designing a new language exceeds the scope of this thesis. Therefore, although this section provides overview of existing approaches, and Chapter 5 discusses additional requirements essential for the language, its design is left for future work.

3.6 Summary

Since enterprise information systems implement business processes in a domain, and maintain large volume of data, it is essential to efficiently deal with cross-cutting concerns. While conventional approaches such as the object-oriented programming are unable to efficiently encapsulate and reuse the concerns, this chapter elaborated several approaches, development models, and technologies addressing the separation of concerns. Unfortunately, only a few recognize impact of business rules on a system. Although the approaches are often specific to a single component, layer, or a technology, or only address some concerns, they determine structure of a system and organization of source code. Subsequently, they reduce development efforts and remove some inconsistencies and error-proneness from a system. Therefore, considering the scope of enterprise systems, a development process may significantly benefit from utilization of such approaches and tools, especially, when a system spends about to 65-90 % of its lifetime in a maintenance phase [55, 80].

Benefits of separation of concerns are demonstrated and discussed in the rest of this thesis. The following chapter introduces the aspect-driven development approach to design of an information system. The approach recognizes cross-cutting concerns as essential components of a system, adjusts its architecture, and proposes a mechanism for efficient separation and automated reuse of concerns in order to address the problems identified in Chapter 1. In addition, since extraction of concerns from an existing system is challenging due to information repetition, possible inconsistencies, and difficult truth resolution, Chapter 6 demonstrates utilization of the approach in common components of a system.

Part II

Contribution

Aspect-driven Development

Chapter 4

Aspect-driven Development

Design and development of contemporary enterprise information systems face various challenges. The separation and efficient reuse of concerns [59, 105], are among the most significant challenges developers deal with (Section 3.2). This chapter introduces a novel aspect-driven development (ADD) approach to design and development of enterprise information systems. This high-level approach to design of a system addresses problems identified in Chapter 1, and utilizes multiple existing approaches to combine their benefits and deliver better concerns management (Objective 1). Subsequently, it reduces manual code duplication, eases maintenance, and increases code quality [43], especially, code complexity, readability, cohesion, and coupling. Implementation of the approach into design of information systems and its evaluation and comparison to conventional approaches is elaborated in Chapters 6 and 7, respectively.

Conventional development lacks the ability to separate and reuse cross-cutting concerns, and tangles them throughout a whole system (Problems 1 and 5), instead. Moreover, as is discussed in Chapters 1 and 2, EISs are designed and developed to manage a business domain implementing business processes, which are constrained by many business rules. Unfortunately, the rules represent another significant cross-cutting concern, which increases the importance of their efficient maintenance (Objective 2). Considering various technologies and programming languages used for implementation of a system (Problem 2), concern reuse becomes difficult. While there exist attempts to overcome this gap as is discussed in Section 3.4, there are no architectures, development approaches, or frameworks providing an efficient mechanism for information systems.

The ADD is an abstract approach to design and development of EISs, and is not bound to any particular language, platform, or architecture. All these are implementation details, which are to be determined later during the implementation of given system. However, for illustration purposes, this work assumes use of the layered architecture and Java programming language since they represent mainstream technologies in conventional development of information systems.

In conclusion, the aspect-driven development approach addresses the separation and reuse of concerns and problems identified at the beginning of this thesis (Objective 1). In order to address the challenge, the approach modifies design and development of a system, its architecture, and representation of concerns, which is elaborated in Sections 4.1 and 4.2. Since the approach utilizes existing approaches (Section 3.3), it is discussed from the perspective of the essential AOP and the MDD in Sections 4.3 and 4.4, respectively. Finally, the impact of the approach on separation and reuse of concerns is discussed in Section 4.5, and evaluation is summarized in Section 4.6.

4.1 Background of the Approach

Since the aspect-driven development approach introduced in this chapter is abstract and high-level, its formal definition might not reveal its broader motivation and context. Therefore, this section provides informal introduction into the approach to emphasize the context and ensure proper understanding of importance of concerns.

Each EIS is built in a business domain to implement its processes, maintain data, and optimize efficiency of related business. Thus a use case (UC), and subsequently a business operation as a step in a scenario, are fundamental elements of a system [67]. The layered architecture leads to the Service Layer and Transaction Script [42] design patterns, which intuitively decompose the domain. Then, each class in the domain layer implements a business process or its part, and each public method of such a class represents a business operation within the process. In consequence, EISs built over the layered architecture are use case centric [67]. Furthermore, the business rules are the most significant, changing, repeated, and tangled concern in EISs, thus it is worth optimizing their representation and management.

During the analytical phase of the development process, use case scenarios are decomposed into business operations. For each operation, there exists a set of preconditions and a set of postconditions defining its assumptions and the expected behavior [67]. Unfortunately, these business rules have to be considered throughout a whole system to deliver reliable and intended behavior. Consider the running example from Chapter 1. There are two following use cases (UCs):

UC 1: Create a new product	**UC 2: List unavailable products**
Preconditions:	Preconditions:
- rule ①: product name is non empty	- rule ④: a current user is an administrator (role Ⓒ)
- rule ②: product price is positive	
- rule ③: product weight is not negative	
- rule ④: a current user is an administrator (role Ⓒ)	
Postconditions:	Postconditions:
- the product was created	- inStock attribute of products is equal to 0
	- rule ⑤: products are not excluded from the catalog

To implement these use cases, the preconditions must be checked in multiple places. For example in the UI to enable or disable an operation, in the server-side of the presentation layer to prevent illegal access and hijacking of the UI, and finally at the input of the domain layer to secure the system and ensure data consistency. This is significant vertical repetition of these rules.

While the postcondition of the UC 1 only describes the expected behavior, the postconditions of the UC 2 must be applied on the output of the domain layer. Although the postconditions generally are not vertically repeated in a system, they are often repeated horizontally due to their occurrence in multiple business operations. Furthermore, they are usually transformed into a query language of the data layer, e.g., SQL, to constrain data already in a persistent storage and reduce application load. This transformation breaks the system architecture because the rules are moved from the domain layer, where they semantically belong, into the data layer to deliver better performance. In consequence, the rules get both horizontally and vertically repeated plus scattered throughout all layers.

The ADD addresses separation of concerns and recognizes business rules as a significant cross-cutting concern due to the use case-centric nature of EISs. The underlying idea of this approach lies in identification, separation, isolated description, and automated runtime transformation and distribution of cross-cutting concerns. Having the concerns separated and inspectable enables their automated transformation and reuse without the need of their manual restatement. It reduces both horizontal and vertical repetition [A.7]. Moreover, untangling the concerns from a complex system structure eases future evolution of a system and mitigates the risks of a human error [A.3].

This approach utilizes fundamental principles of several existing approaches to tackle issues introduced by cross-cutting concerns. More specifically, to separate those concerns, transform them, and then weave back together. For example, the MDD utilizes several levels of abstraction to generate the code via forward transformation [63]. The AOP provides a mechanism to separate concerns within the OOP and introduces the concern (aspect) weaving [62]. The CDD recognizes cross-cutting concerns as independent first-class citizens of development, emphasizes their importance, and focuses on their separation, efficient description, and reuse [68]. The DSM accelerates the development through multiple DSLs tailored for a particular domain [58]. All these principles significantly improve development and inspired the ADD.

Similarly like the CDD, the ADD perceives cross-cutting concerns as independent reusable units and individual axes in a multidimensional space [A.8]. There exist both static and dynamic concerns. Static concerns are immutable and stable, thus it is sufficient to process these concerns just once, e.g., at compile time, because they do not change over time. The domain model, its fields, and their types are examples of the static concerns as well as UI elements used for data visualization because they are derived from the domain model. On the contrary, the dynamic concerns consider runtime information such as a current user, an invoked business operation, current timestamp, or an application state [A.7] and thus must be evaluated at runtime. Business rules with validation constraints, integrity checks, and access control policy are examples of a dynamic concern. In consequence, the ADD perceives execution of a business operation as a single point in the multidimensional space [A.8]. The point is determined by the current execution context, which puts high demands on efficient processing, runtime evaluation, and the final composition of concerns.

The repository is a novel concept introduced by the ADD and inspired by the MDD. This component is an abstract model, possibly platform-independent, and responsible for maintenance of all concerns. It represents a single point of truth, i.e., a single focal point, in the application [A.8] (Problem 3). All concerns in the repository are described in inspectable DSLs rather than in a general programming language, which follows the domain-specific modeling. DSLs are more efficient in describing domain-specific logic, and tailored for a particular domain, i.e., a concern, while relaxing stress on generality. It significantly reduces development and maintenance efforts, increases readability, and enables domain experts to directly participate in the system development [72] (Problem 5). Furthermore, with a proper design of a DSL, the concern representation is both efficient and inspectable and ready for further transformation [A.7].

Having those concerns separated and described in the repository, the ADD uses runtime weaving to compose cross-cutting concerns (Problem 4). The concerns weaving mechanism is the aspect weaving from the AOP, which accepts decomposed concerns in various languages. Then, it transforms them into the target platform (Problem 2) and combines them together according to the composition rules [A.3] (Problem 1).

The layered architecture consists of several components and layers, each considering a different set of concerns, implementing different logic, and having different responsibility. The ADD reflects this differences in providing multiple weavers, each designed for a single component [A.13]. For example, there exists one concerns weaver for the client-side of the UI and another weaver for the data layer easing the querying of a persistent storage. This focus on a purpose simplifies composition of concerns, their intended use, and production of component-specific code.

The resulting code is always context-aware because weavers accept both static and dynamic concerns together with an execution context [5, A.1]. In consequence, the ADD produces only combinations of the concerns that are actually used, therefore it is more flexible and fights the exponential growth of combinations, which is a significant limitation of GP[1] [25]. Furthermore, runtime code generation enables more advanced optimization of the code for a particular combination of concerns [62].

[1] A single combination of concerns is considered a state in terms of the GP. This relates to a single execution in the ADD, which is a combination of static and dynamic concerns.

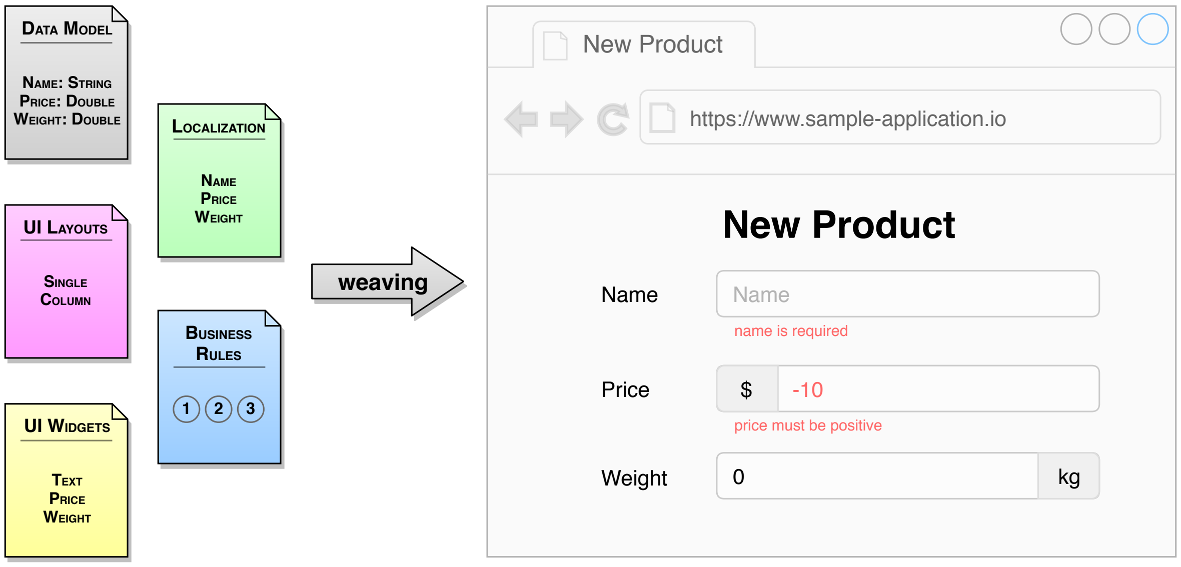


Figure 4.1: Concern weaving into the UI form

To conclude the given example, consider implementation with the ADD. First, all business rules are represented in the repository using suitable DSLs. Second, also other concerns such as UI layouts and UI widgets [A.3, A.5] are located in the repository. Then, those concerns are only referenced, e.g., by a name, from places, where they must be considered. Finally, runtime weavers inspect the repository and compose the linked concerns to context-aware code including a persistent storage querying [A.7] and the UI [A.1]. The difference between the conventional development and the ADD lies in the automated transformation and distribution of concerns, which eliminates their manual duplication. It also enables use of DSLs, which brings many benefits [72].

For better illustration, consider this implementation of the UC 1. The UI weaver accepts the name of the rules ①, ②, ③, a form layout, and a class as a typed model of the input form. Then, it produces HTML with Javascript according to the fields in the class as is illustrated in Figure 4.1. These fields are protected by Javascript validators implementing the rules and the widgets are picked based on the composition rules matching the situation [A.3]. Next, the server-side UI weaver accepts the model of a form and validates it with the rules or their subset to avoid UI hijacking. Finally, the domain layer weaver accepts business rules and a method in the domain layer implementing a business operation. On a request, the weaver validates the incoming `Product` instance and if is valid, then it invokes the internal method. The implementation is further elaborated in Chapter 6.

In conclusion, the aspect-driven development approach recognizes and emphasizes the role of cross-cutting concerns in EISs and modifies both design and development of a system to provide more efficient mechanism for their representation and reuse (Objective 1). The approach describes concerns in domain-specific languages (Problem 2) and maintains them separated (Problem 1) in the single point of truth (Problem 3), which removes both manual repetitions and the need for manual synchronization of multiple places (Problem 5). Instead, the ADD utilizes aspect weaving from the AOP to compose those concerns back together at runtime, which enables context-aware customization of a system (Problem 4).

4.2 The Architecture

The ADD decomposes a system into several additional components comparing to the conventional development. While the background idea is informally described in the previous section, this section provides more formal description, puts down important definitions, and provides a schema of the modified layered architecture.

The separation of concerns is the essential concept of the ADD, and as a result, the ADD identifies more concerns comparing to the convention development. Newly recognized concerns include UI widgets, UI layouts, and most importantly business rules (Objective 2). The importance of business rules rises from the use case-centric nature of EISs [67], where each use case scenario fulfills the purpose of a system. A business operation[2], which is a step in the scenario, defines assumptions to be valid before its execution (the preconditions), and assumptions to be valid after its execution (the postconditions). Let the ADD define a set of all business operation assumptions to be a business context (Definition 4.1). Since use cases are derived from processes in the business domain, the business context is also transitively defined by the domain.

> **Definition 4.1** *A business context is a set of parameterized preconditions and postconditions of an operation with a business value [A.7].*

The separated concerns are described in suitable DSLs [62, 110]. The ADD introduces a new component to store descriptions of the concerns, the *Concern Repository* (Definition 4.2). This component is parallel to the existing architecture because it aggregates cross-cutting concerns and thus is accessed by all other components and layers, which is illustrated in Figure 4.2. The design, the structure, and the implementation of the repository may differ based on the implementation of the ADD. However, the repository must implement several responsibilities:

- store multiple concerns,
- support multiple different DSLs,
- provide a key-value storage for each concern,
- and support concern retrieval for the subsequent inspection and transformation.

Besides these essential requirements, the repository optionally provides additional features such as a cache storage for preprocessed concerns or support for distributed

[2] The business-related terminology explained in more detail in Section 2.1.

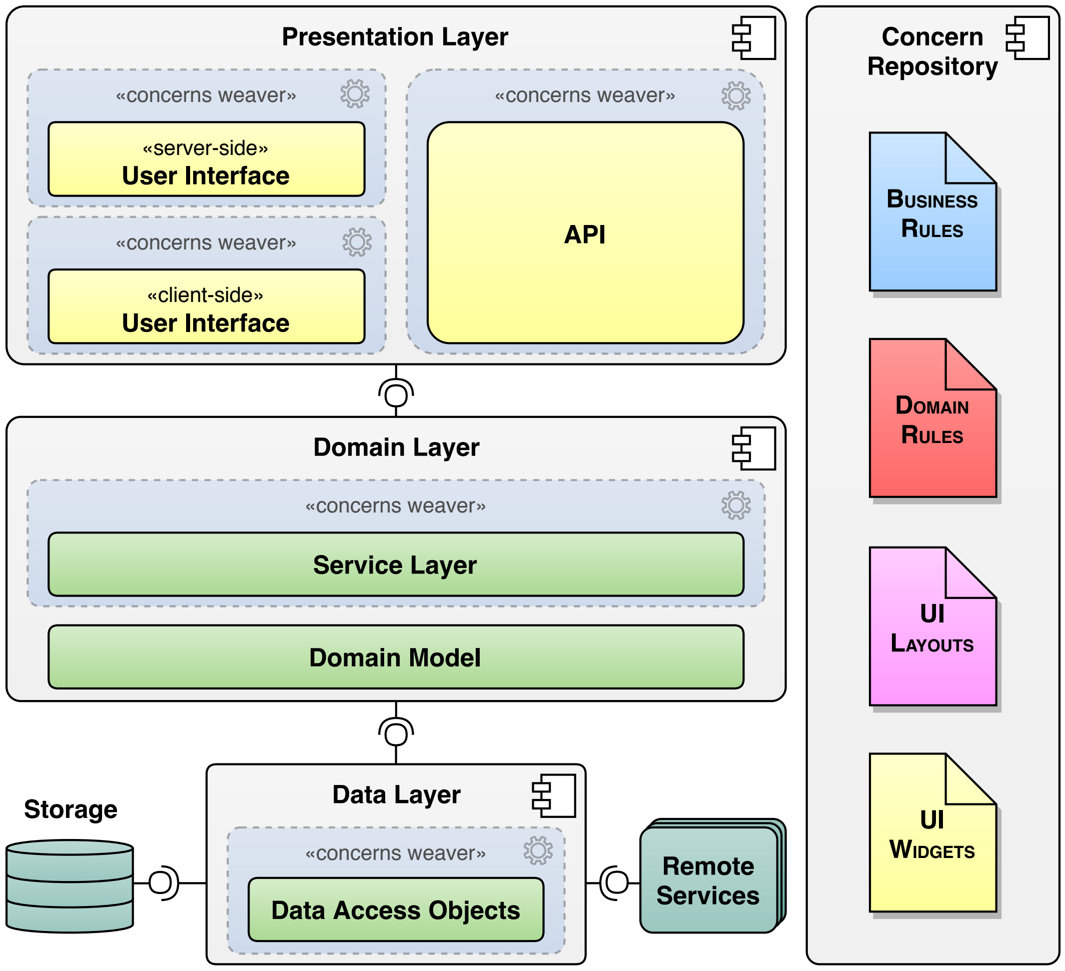


Figure 4.2: The layered architecture with the aspect-driven development

platform-specific concerns [A.1], which is discussed in Chapter 6. However, these optional improvements are implementation details and do not affect the overall concept.

> **Definition 4.2** *The concern repository is a cross-cutting component representing the single point of truth in a system. It is a key-value storage accepting concerns in multiple DSLs for inspection, transformation, and runtime distribution.*

The separation of concerns results in number of independent concerns and the code being enriched. In consequence, the source code is not actually executable and requires further processing instead. To deliver executable code, the ADD utilizes *the concern weavers* (Definition 4.3). The weaver is a component accepting the code to be enriched and concerns to be applied, and producing executable code. To compose concerns and the code, the weaver uses composition rules. For example, consider code of the user interface in the presentation layer and code querying a persistent storage in the data layer. Although both components consider business rules, the composition rules are very distinct. There are differences in used programming languages, frameworks, and intentions. To simplify the implementation of weavers, a weaver is designed for a single component, accepts specific concerns, and composition rules, which fit requirements of the intended use. As a result, each EIS deal with multiple weavers.

Definition 4.3 *The concerns weaver is a component implementing aspect weaving from the aspect-oriented programming to weave concerns and code together at runtime and produce executable concern-aware code.*

Note *The concerns weaver accepts concerns in various DSLs as well as GPLs, each concern described in the most suitable language, for greater efficiency and quality of the resulting code. The weaver runs at runtime to also accept dynamic concerns, and delivers context-aware executable code. Usually, a single concerns weaver is designed for a single component to inspect and transform only selected subset of DSLs and produce a component-specific code.*

The composition and the distribution of concerns are provided by the weavers at runtime, which also makes the dynamic concerns available. There exist an instance of a current user, the request, and the invoked business operation. Dynamic components have significant impact on composition of concerns, thus the weaver is unable to properly compose the concerns at compile time. For example, user's role or a geographical location may significantly impact the validation or even available fields [17].

Concerns weavers are usually the interceptors. They wrap the original component and its API and intercept all incoming invocations as is illustrated in Figure 4.2. When invoked, the weaver collects all static and dynamic concerns, takes the underlying code (if any[3]) and produces the executable code [A.8]. There are many possibilities of what the concerns weaver may produce. For example, it may generate a form in the UI [17, A.3, A.5], validate the input in the domain layer [A.15], or construct a query into a persistent storage [A.7]. The actual code depends on the focus of the weaver and responsibility of the wrapped component. Examples of possible weavers, their execution, and resulting code are discussed and demonstrated in Chapter 6.

Majority of all concerns are dynamic concerns. The properties such as a current user, a geographical location, a timestamp, or a used device optionally impact validation, available actions, a UI structure, or a UI layout. While there exist various dynamic concerns, the ADD groups them up into contexts[4]. Their structure is illus-

[3] Incoming requests into the domain layer invoke a business operation, which is wrapped [A.15]. Considering the UI inside the presentation layer, there may not exist any code to be wrapped and the concerns weaver may generate it from scratch, e.g., an input form [A.5].

[4] Dynamic concerns vary for each request, operation, and in time, thus create a current *context* for current code execution, which gives them the name.

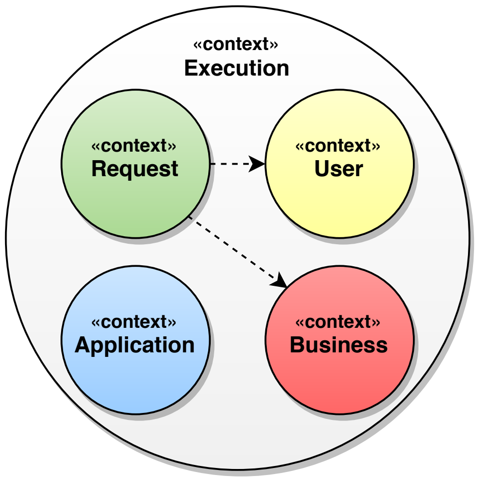


Figure 4.3: The contexts in the aspect-driven development

trated in Figure 4.3. The *execution context* is a top-level context encapsulating all other contexts and clarifying the terminology. It encapsulates all existing dynamic concerns and is accepted by weavers as the runtime input component.

> **Definition 4.4** *The execution context is a complex structure utilized during concerns weaving and encapulating all dynamic concerns [A.7].*

A current user is an important part of the execution context. The user affects business rules through defined roles, and profile settings may impact other aspects of a system. The ADD recognizes this concern to be the *user context* [A.7]. It encapsulates all properties attached to a user including language preferences, roles, and privileges.

> **Definition 4.5** *The user context defines a current user, his username, access roles, privileges, and profile settings [A.7].*

While a user must be authenticated in a system to determine the context and there are many implementations of pairing a user with a request, the ADD is agnostic to the implementation. However, the authentication defines a relation between the user and the request. The *request context* encapsulates all properties defined by a single request into a system. Besides the current user, the context usually includes an IP address, a timestamp, a geographical zone, a protocol, a web browser, and a user's device.

> **Definition 4.6** *The request context encapsulates properties specific to a current request including a timestamp, user's IP address, a geographical location, a user-agent, and a screen size [A.7].*

Since EISs are use case-centric, each request targets a single business operation. The operation and its preconditions and postconditions compose the *business context* (Definition 4.1), another dynamic component in the execution context.

Finally, the last member of the execution context is the *application context* (Definition 4.7), which represents a state of given system, especially system-wide variables. For example, an e-commerce system can be locked for changes due to upcoming maintenance and current VAT rate is also a system-wide variable. All these variables are collected in the application context and considered as binding of variables in business rules. See Chapter 5 for more details.

> **Definition 4.7** *The application context encapsulates global system-wide variables and their values at a given point in time [A.7].*

In conclusion, the essential and the most important idea of the ADD is the extended understanding and the separation of both static and dynamic concerns. With static concerns located in the repository and dynamic concerns provided at runtime, weavers produce executable context-aware code specific to a particular component. Although this sections discusses the ADD with the layered architecture, the approach is agnostic to any implementation details and does not depend on any technology, language, or architecture. The ADD itself extends any conventional architecture, adds the additional component, wraps existing components, and extends understanding of cross-cutting concerns. It is a general development approach applicable to any conventional approach, although the benefits and limitations may significantly vary [A.10].

4.3 Definition in Terms of the AOP

While the ADD utilizes multiple existing approaches, the aspect-oriented programming (AOP) is its essential component. The ADD recognizes cross-cutting concerns as aspects and composes them together via aspect weaving. In consequence, there exist tight coupling between the ADD and the AOP, which this section defines in a formal specification. In addition, formal mapping into the AOP provides a hint to implementation of the approach.

The AOP is a technique of automatic programming targeting the separation of concerns, designed to decompose cross-cutting concerns, and then weave heterogeneous sources together. It is a convenient way to process all descriptions of distinct concerns and weave them into source code. These are perfect assumptions to fulfill Objective 1 of this work, therefore the ADD advances this underlying idea.

The concerns in EISs cross-cut both horizontally and vertically through all layers and components. Each component is designed for a specific purpose and possibly uses different technologies. In consequence, there must exist an AOP specification for each component as is further elaborated in Chapter 6. However, here follows a brief introduction into the AOP specification. It clarifies the terminology and boundaries of each term. For better illustration, the specification uses the running example and more detailed examples and deployment of the ADD follow in the next chapters.

The AOP comes with multiple terms, which must be defined in order to properly formalize the ADD in terms of the AOP. The mapping definition must include: *aspects*, *joinpoints*, *pointcuts*, an *advice*, an *aspect language* and finally an *aspect weaver*.

The aspects overlap with the concerns in the ADD terminology. They involve both static and dynamic cross-cutting concerns in the system, which tend to manual restatement, but are separated, described independently, and located in the repository.

For example to implement the UC 1 from Section 4.1, there exists a UI form illustrated in Figure 4.1. Such a form reflects the structure of a model (class `Product`), provides presentation for each of its fields (UI widgets), but also considers all dynamic concerns including business rules to implement security, validate the input, and select layout according to a user's device. All these concerns, which influence resulting presentation and implementation of the form, are aspects in EISs in terms of the AOP.

The joinpoints in the ADD are locations in source code (static joinpoints) and execution flow (dynamic joinpoints), where the concerns must be considered, i.e., the code or the flow modified. The ADD defines several types of joinpoints depending on a concern and a target component. They include entry points and exit points of business operations, methods for data retrieval, UI buttons, and a lifecycle of UI forms.

For example, business rules may be applied in the domain layer as input validation in business operations and thus the joinpoints are entry points of the operations. While in the presentation layer, business rules are considered as input validation in forms, and thus the joinpoints are inside the lifecycle of a form.

The advice transforms a concern into the target domain of a specific component and technology. The kind of the transformation differs per component, because the use of the concern may significantly vary. An advice optionally provides an integration template expecting a transformed concern and defining new joinpoints for subsequent concerns. These are then filled by the aspect weaver according to composition rules.

For example, consider input validation in the domain layer. The rules are transformed into an executable form to be bound with the input and the execution context, and then evaluated. On the contrary, in the UI in the presentation layer, the rules are decomposed and transformed into client-side, e.g., scripting, language and attached to fields of a form and evaluated later when needed. One example of a complex advice is a UI layout. It defines a template in a language of the presentation layer and defines new joinpoints to inject presentations of form fields and their labels.

The pointcuts select a subset of joinpoints to indicate, where to apply an advice. The advice contains a set of pointcuts defining places of use. Considering multiple specialized advices matching a single static joinpoint, a pointcut can query an execution context for runtime conditions, to match only the cases of intended use.

For example, consider business rules. Pointcuts for the advice implementing input validation in the domain layer match only entry points of business operations. On the contrary, consider an advice implementing a UI widget with an input for weight designed for small screens. Such pointcuts select places inside a UI layout, where weight is expected when an execution context contains the small screen size flag.

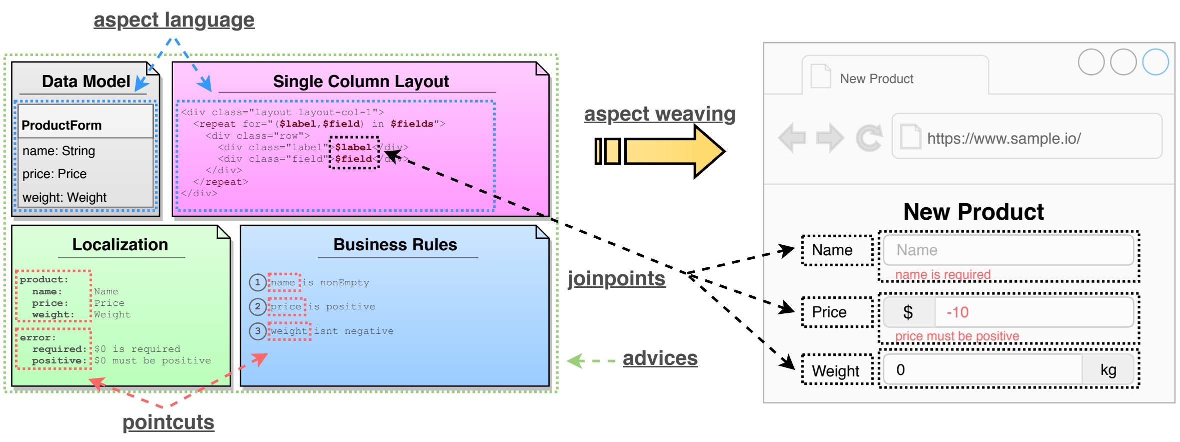
The aspect language is the language used for implementation of advices. Considering the underlying idea of the ADD and the benefits of the domain-specific modeling, there is no specific aspect language. As there exist multiple aspects, there also exist multiple languages, each tailored for a particular domain, which is an efficient approach [110]. Unfortunately, while DSLs introduce great benefits, they also have significant limitations, which are discussed later in this thesis.

For example, while a UI layout as well as UI widgets in web-based EISs are efficiently describable in HTML [A.3], localization description is more efficient in some key-value language such as YAML [7]. UML [96] and general-purpose languages such as Java are well suited for classes, types, and attributes of a domain model, and business rules are efficiently represented in a language such as JBoss Drools [11].

The aspect weaver can be considered as a concerns compositor. It is responsible for weaving advices together according to composition rules, production of executable code, and modification of the target location with new functionality. A weaver accepts multiple advices in various DSLs and the target execution point, and then produces component-specific code replacing the original functionality. It is essential that the weaver is tailored for the intended use to support proper technologies, concerns, and languages as there exist many different execution points, each component uses different language, and there is no unified aspect language. In consequence, a single EIS following the ADD can utilize multiple weavers, each tailored for a different component.

For example, an aspect weaver composing forms in a UI accepts localization, a form model, UI widgets, a UI layout, and business rules and then produces a form to be displayed in a web browser. With runtime context-aware composition, the layout fits the screen size, the widgets are picked with respect to data types, localization, and the screen size, and the validation rules consider user's context and privileges.

Input validation in the domain layer is another example of the use of an aspect weaver. The weaver accepts both business and execution contexts and the business operation. Then it registers the validation code constructed from the rules in the business context at the entry point of the operation. When invoked, the code either throws an exception when the input is not valid or passes the execution into the operation.



```
<div class="layout layout-col-1">
  <repeat for="($label,$field) in $fields">
    <div class="row">
      <div class="label">$label</div>
      <div class="field">$field</div>
    </div>
  </repeat>
</div>
```

```
product:
  name:     Name
  price:    Price
  weight:   Weight

error:
  required: $0 is required
  positive: $0 must be positive
```

```
1 name is nonEmpty
2 price is positive
3 weight isnt negative
```

Figure 4.4: The AOP components in the UI of the running example

To summarize the specification, Figure 4.4 provides a complex example highlighting all AOP components. The example shows the aspects participating in the construction of the form from UC 1. Concerns from the example include a model of the form, the business rules from the UC specification, the single column layout matching the screen size, and the localization. All these concerns are also aspects in terms of the AOP. UI widgets are omitted in order to reduce the size of the figure. This example highlights the advices, i.e., the concerns to be transformed, pointcuts addressing the joinpoints to apply the advices, and jointpoints introduced by the upper-level advice (the layout). These jointpoints are used in later iterations of the aspect weaver to inject the UI widgets. Notice that each aspect uses a different aspect language. Finally, the process of concerns composition, i.e., aspect weaving, is implemented by an aspect weaver. However, full implementation of UI composition is discussed in Chapter 6.

In conclusion, this section shows the importance and the role of the AOP within the ADD. It is obvious that the ADD extends the AOP, while it adds several concepts from other approaches. As examples suggest, there exist various components benefiting from this novel approach. Chapter 6 focuses on the implementation of the ADD into design of common components of the layered architecture and introduces further and more detailed mapping between the ADD and the AOP.

4.4 Perspective of the Model-driven Development

Similarly, while the ADD extends the AOP to recognize concerns as aspects and compose them together via aspect weaving, it is significantly inspired by the MDD. The common idea lies in maintenance of various models on multiple levels of abstraction, which is utilized during the aspect weaving. Therefore, this section discusses the ADD approach from the perspective of the MDD to identify common concepts and differences, and help to comprehend the aspect-driven development in a broader context.

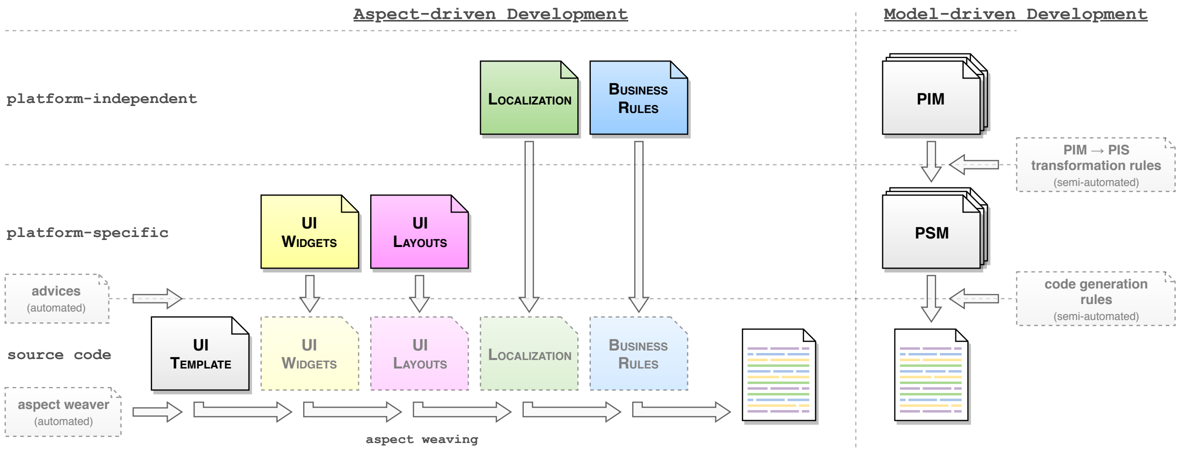


Figure 4.5: Model transformation process with the ADD and the MDD

The MDD utilizes several models on different levels of abstraction to minimize the manual information repetition and duplication. Then it uses model transformation as a forward engineering method and code generation tool. However, the transformation process produces more specific but incomplete models as those more abstract models do not contain enough information. It leaves some efforts for developers to manually fill in blank spaces in new more specific models [63]. This transformation process usually occurs during development as it is necessary to fill in the gaps, i.e., the models are transformed before compile time. Unfortunately, the MDD often fails when comes to backward change propagation because the abstract models are unable to represent more specific information. That means, when the more specific model is changed, then regeneration of the specific model erases manual refinements from it.

On the contrary, while the ADD also uses models on several levels of abstraction[5], they are complete and do not leave any efforts to developers. The forward transformation process is fully automated and happens at both compile time and runtime[6], which is an important difference. In consequence, the produced models are not intended for manual changes so there is no need for backward change propagation.

Another significant difference lies in the model transformation. The MDD accepts abstract models and then sequentially performs vertical transformation to produce more specific models such as platform-independent, platform-specific, and finally the source code. Each level of models has more tangled concerns to address the requirements, which results in extensively tangled source code. This process is illustrated in Figure 4.5. On the contrary, the ADD utilizes the *vertical* forward transformation to transform concerns into proper platforms and components (advices) and then *horizontally* composes all concerns within a single component together via the aspect

[5] The repository is a model of the concerns. However, there are possibly both platform-independent and platform-specific concerns, which is discussed in the subsequent chapters. Code to be enriched by the concerns is another model, often a model of business logic.

[6] Some concerns can be transformed at compile time or during system initialization, but others need the execution context and thus must be transformed at runtime.

weaver [A.7]. While the vertical transformation is responsible for concern transformation, the horizontal transformation deals with concern tangling. This distribution of responsibilities reduces complexity of both transformation processes and eases their automation. Furthermore, while the vertical transformation can be precomputed at compile time or during system initialization, the horizontal transformation requires the execution context to deliver context-aware code thus must run at runtime.

Consider the example in UC 1. There exist business rules and localization described in platform-independent DSLs, and UI layouts and widgets described in a DSL specific for web browsers, i.e., platform-specific. Then the rules and the localization are transformed into technologies of the UI (vertical transformation). Finally, the UI template expecting the form and all these concerns are weaved together considering the execution context (horizontal transformation). This process is illustrated in Figure 4.5.

Completeness of the model in the ADD approach is an essential benefit. In case of model evolution, there is no need to manually update any subsequent models, they are all refreshed in the next compilation/system initialization cycle. In conclusion, while the ADD is inspired by the MDD and utilizes similar techniques and transformation rules, the model completeness and horizontal transformations make a significant difference in the overall approach efficiency.

4.5 Reuse of Concerns

Since the approach addresses the separation of concerns, this section briefly discusses its impact on concerns reuse. However, detailed implementation of the approach into design of common components and examples of concerns reuse are provided in Chapter 6. While the ADD significantly modifies architecture and development of EISs, changes in design mostly impact components, classes, and UIs instead, which greatly benefit from reuse of concerns. Having these concerns isolated and described in the repository opens possibilities of their automated mining, transformation, and integration to save development and maintenance efforts.

Mining is an extraction of information from the concerns in the repository. There exist reverse engineering techniques extracting information from an EIS, getting overview of a system, analyzing its structure and behavior, or just computing chosen statistics. Having the concerns separated significantly reduces the mining complexity as the concerns are not tangled but are represented in an inspection-friendly form instead. For example, business rules can be validated by domain experts, or transformed into a formal specification and verified against requirements. Similarly, generation of actual documentation via reverse engineering of the structure, modules, classes, operations, and business rules is usually a very complex task [100], but the ADD generates the documentation through simple inspection and transformation of the concerns [A.2].

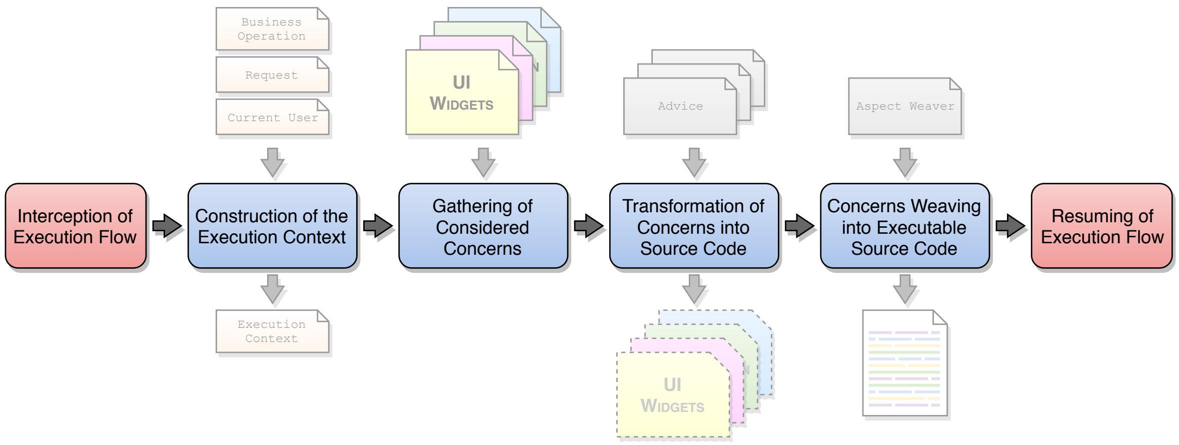

Business Operation
Request
Current User
UI Widgets
Advice
Aspect Weaver
Interception of Execution Flow
Construction of the Execution Context
Gathering of Considered Concerns
Transformation of Concerns into Source Code
Concerns Weaving into Executable Source Code
Resuming of Execution Flow
Execution Context
UI Widgets

Figure 4.6: Concerns weaving process

Transformation and integration is the other possibility of concerns reuse. As the previous sections define, the ADD uses the forward transformation to transform the concerns into other languages and technologies and to weave them together. That enables automated reuse of information in multiple places without any need of manual information duplication. The impact on maintenance efforts is obvious.

However, regardless of the component, the integration process follows a same pattern, which is illustrated in Figure 4.6. First, consider the context of use. It defines the execution context, i.e., a current user, a request, and a business operation. Second, pass the flow to the aspect weaver with the target place and all parameters. Third, considering the context, execute given advices to get the context-aware concerns in a platform-specific format. Fourth, the aspect weaver weaves all concerns into the target location, and finally, passes the flow to the generated code.

In conclusion, the separation of concerns and having concerns in an inspectable form opens wide range of possibilities of their further reuse. Although, particular implementation of the ADD and possible reuse or concerns significantly depend on the used architecture and technologies because each comes with different components and support of the AOP. For complete illustration, Chapter 6 presents several implementations of the ADD into design of an EIS with the layered architecture.

4.6 Benefits, Limitations, and Summary

The aspect-driven development emphasizes the separation of concerns, their independence, efficient description, and automated transformation. It deploys the repository as a new component parallel to the architecture to represent a single point of truth in a system, and maintain the concerns described independently in DSLs. This section sums up qualities, benefits, and limitations of the approach to conclude this chapter.

The main contribution lies in automatic concerns reuse, which includes runtime weaving. While it significantly reduces manual information repetition, the concerns are automatically distributed throughout the system by aspect weavers (Problem 1). This leads to more efficient development and maintenance of a system, as well as it mitigates the risk of human error, comparing to manually repeated and tangled concerns (Problem 5). Furthermore, the concern distribution can be carried out across different platforms, which helps to use various technologies for individual modules (Problem 2), while it preserves the single point of truth and concerns reuse (Problem 3). Design of aspect weavers enables dynamic pointcuts and concerns. Both consider contextual information such as user's identity, privileges, IP address, current application state, and server load, and use them to select proper joinpoints to weave in. However, it requires runtime weaving and efficient concerns transformation (Problem 4).

Use of DSLs enables responsibility delegation to domain experts and subsequently better work distribution in a team. Having the repository as a single point of truth eases development and maintenance. Its design simplifies description of concerns, reduces error-proneness via isolation of concerns, and simplifies their testability. Furthermore, independent concern description allows their reuse among projects, for example, UI widgets, UI layouts, and security policies are not project-specific.

Extraction of these concerns out of the code base is a major simplification of the code. Usually, the code no longer needs to focus on preconditions, layouts, and other concerns, and can focus on business logic and behavior instead. There is also major code reduction via removal of duplicate code implementing the repeated concerns. As a consequence, it reduces maintenance efforts and error-proneness as there are no longer multiple places to update and the code base is reduced.

Disadvantages include approach complexity, which has the most significant impact on overall efficiency. Although the idea is simple and straightforward, there are many DSLs and multiple separated concerns, which increase mental complexity. Developers must get used to the novel architecture and must learn the languages, and domain experts must be trained as well. In consequence, there is a high mental barrier.

Besides the mental complexity, there exists significant initial overhead reducing the approach efficiency on small projects. Although the languages and their components are reusable among projects[7], their initial implementation is a big challenge and requires significant efforts. All these languages must be designed including their compilers/interpreters and possibly editors. Next, there are complex aspect weavers accepting all independent concerns in these DSLs and producing various context-aware components such as the domain layer, and the UI [A.13, A.8]. Unfortunately, an aspect weaver is specific to a tuple of input concerns, an output component, and a technology, thus the number of weavers grows with the number of target components and technologies. With runtime weaving, performance is critical and increases the efforts.

[7] The DSLs for business rules, UI layouts, widgets, etc. are not project-specific and can be reused in other project within the same domain and possibly also outside of it. The same stands for all components including the weavers as long as the technologies are same.

Comparing to the existing approaches, the ADD suffers from the overall complexity. On the other hand, contrary to the MDD, it does not suffer from the issue of backward propagation of changes to more abstract models and is tailored to deal with cross-cutting concerns. The benefits of concern-driven development and the ADD are similar. Both approaches focus on the cross-cutting concerns and deal with them via DSLs and automated transformation. The ADD also benefits from the context-aware runtime weaving, while the CDD prefers composition of concerns during implementation of a system. Comparing to the AOP, the ADD is more focused and provides a specific solution to EISs, while AOP is a general technique to deal with cross-cutting concerns in software engineering. The benefits of the ADD over the domain-specific modeling lie in more universal and possibly project-independent DSLs and other tools. While the DSM always tailors languages for a particular domain and implements all supporting tools, the ADD is more abstract and the DSLs do not have to be tailored to a particular business domain, they are tailored for a concern instead, which makes them more transferable across projects. Finally, the benefit of the ADD in comparison to the GP lies in the runtime context-aware weaving in trade of more complex tools.

In summary, while conventional development approaches fail in addressing cross-cutting concerns and tend to manual concerns repetition, the aspect-driven development approach introduces an alternative design of enterprise information systems. The approach is efficient for large enterprise applications, and all its benefits arise from decomposition of the multidimensional space of the concerns, and their independent description in efficient domain-specific languages (Objective 1). While the separation of concerns is an essential part the approach, it recognizes business rules among the significant concerns within a system (Objective 2). Considering a use case-centric system, business processes and their operations are constrained by many business rules, which cross-cut throughout the whole system. Although, recognition of this concern and its separation into the repository puts high demands on a language for business rules description. Therefore, Chapter 5 discusses the impact of the ADD on the language, and identifies requirements and potential benefits of having this concern separated.

Although the aspect-driven development approach is inspired by multiple existing approaches and combines their benefits, it suffers from significant limitations. It deals with cross-cutting concerns, efficiently addresses Objectives 1 and 2 and the problems identified in Chapter 1, reduces error-proneness, and development and maintenance efforts, but the initial overhead and complexity of supporting tools introduce a significant initial barrier. In addition, the use of multiple domain-specific languages represents another mental barrier. In conclusion, while the approach promises great benefits and addresses significant issues, the limitations reduces its overall efficiency and reduces the area of use to large long-running systems.

Chapter 5

Representation of Business Rules

The aspect-driven development approach introduced in Chapter 4 recognizes various cross-cutting concerns as important components of EISs (Objective 1), and modifies design of a system to separate the concerns and enable their automated reuse. As is stated in Chapters 1 and 2, EISs are use case-centric applications focused on implementation of business processes consisting of business operations, which define preconditions, and guarantee postconditions. In consequence, EISs are significantly constrained by various business rules, which belong among cross-cutting concerns, since they are considered throughout the whole system, from the data layer though the domain layer to the UI in the presentation layer. Furthermore, since EISs exist to implement business logic, business rules are an essential concern. Therefore, their efficient representation is crucial for development efficiency, as they tend to manual repetitions, which eventually results in inconsistencies and errors in source code. While design and implementation of efficient representation of business rules is a great challenge itself and is not in the scope of this thesis, this chapter defines requirements for the representation of the rules in the ADD to address Objective 2, provides overview of further utilization of already maintained rules, and lays foundations for further research.

An EIS implements business processes[1] of a business domain to ease domain maintenance and achieve a greater objective. Each process consists of several steps, each implementing a partial objective. The steps are linked into a graph with guarded transitions, which correlates with stateful Event-Condition-Action systems [64]. For example, a guard can check an output of a previous step, consider current time, or a geographical zone of a user. The example of a business process implementing a creation of a new product (UC 1) in the running example is illustrated in Figure 5.1. A step with varying output can be the input from the user. Deeper elaboration and change of

[1] Both a business process and a use case describe same reality but differ in perspectives and level of abstraction. While a process considers business area perspective and emphasizes a flow of activities, a use case considers a technology and describes actor's interaction with a system in a scenario [67].

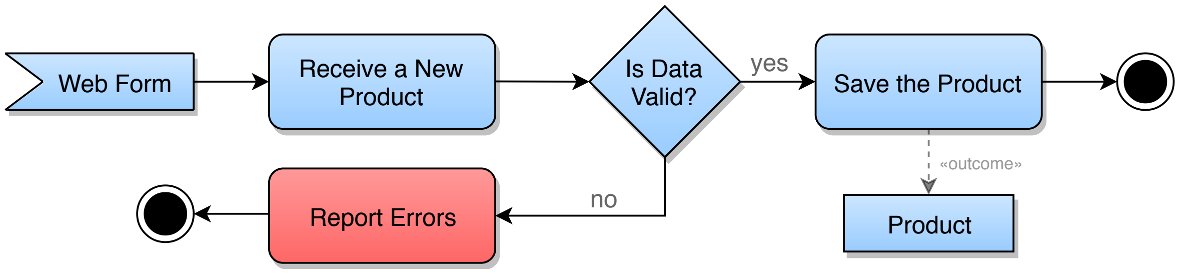


Figure 5.1: The process of product insertion in the running example

perspective transforms a business process into a use case. Consider this full description UC 1 with a main successful scenario and also with alternative paths [67]:

Use Case Section	**Description**
Title	UC 1: Create a new product
Summary	This UC describes creation of a new product
Primary Actor	Administrator
Preconditions	An administrator is logged into a system
Main Flow (Successful Scenario)	
1. Administrator opens a form for product creation 2. The system requests administrator to provide details 3. Administrator fills the form with the details 4. Administrator submits the form 5. The system validates the received data 6. The system creates the product 7. The system notifies administrator about product creation	
Alternate Flow	If the data validated at the step 5 of the main flow is invalid then: 1. The system notifies administrator about invalid data 2. The UC resumes at the step 2 of the main flow
Exception Flow	If administrator requests to interrupt creation of a new product at the step 3 of the main flow: 1. Administrator requests to cancel the process 2. The system cancels the process 3. The UC ends
Postconditions	Main flow: The new product is created Exception flow: The process is canceled

From the main successful scenario it is obvious that each step has its own position in the process and defines its own assumptions (preconditions) validating a context of use and postconditions guaranteeing a result. In consequence, a step is defined through its preconditions, invariants, and postcondition [67].

Consider the terminology used in this thesis (Chapter 2). Steps of a business process are business operations and their context of use is the execution context (Definition 4.4), which encapsulates all runtime concerns specific for current execution. A business context is one of its components, determines a business operation, and encapsulates a set of related preconditions and a set of related postconditions (Definition 4.1). The examples of such a context are the UCs 1 and 2 from the previous chapter. The ADD emphasizes separation of concerns and considers business rules and subsequently business context to be among the concerns to separate. In consequence, it is essential for the ADD implementation to efficiently express a business context, enable its inspection, transformation, and further reuse. Therefore, this chapter analyses requirements for business context representation, and subsequent reuse of business rules already represented in a system. Finally, considering the results from Section 3.5, this chapter proposes an example of a potential domain-specific language for business rules representation suitable for the aspect-driven development approach.

5.1 Analysis and Requirements

In order to separate business rules, and allow their efficient management, and automated reuse, the representation must address complex requirements defined by the role of the rules in a system, and a mechanism of the aspect-driven development.

Language selection is the top level requirement. It affects maintenance as well as inspection and transformation. Poorly selected language reduces efficiency and might not address advanced requirements, therefore some languages do not even qualify for the use with the ADD. Considering the separation of concerns, the model-driven development, the concern-driven development, and the aspect-oriented programming, then domain-specific languages are the first choice [47]. With focus narrowed to a business domain, they enable involvement of domain experts into the development process, while increase expressiveness and enable language customization to fit relevant needs. In consequence, a DSL addressing these requirements must either exist or be tailored.

Use case-centric understanding of a system emphasizes the role of a business context. Considering a business operation as a step within a process and as the target of each request, then the context must be well described. The language addresses the context as a top-level element consisting of a set of preconditions, and a set of postconditions. The name of the context links it to the operation and enables its referencing in source code. For example, consider the *insertProduct* operation from the UC 1 inserting a new product. Use of this context in source code of the domain layer is illustrated in Listing 5.1.

Conditions are fundamental elements of a context as they construct a set of preconditions and a set of postconditions. Many existing approaches deal with both assumptions and consecutive actions, which are executed when the assumptions are met. However, Listing 5.1: Context reference in source code in the domain layer

```
@BusinessContext("Insert a new product")
Product insertProduct(Product product, User actor) {
  // the execution context including the product is already
     validated
  // insertion of the product into a persistent storage
}
```

the ADD understands business rules as preconditions, invariants, and postconditions in use case scenarios. The subsequent actions are understood as business logic, which differs according to the execution context. For example, in the domain layer, the logic is actually implemented in a general-purpose language, but in the UI, it only enables or disables a button to indicate that the operation is available. In consequence, the rules in the ADD are only boolean expressions always evaluated to either true or false.

Variables and constants are further components supported by the language. The variables are bound to the current execution context, which includes the current user, the request, and privileges, but also parameters injected from source code. Consider Listing 5.1, both parameters *product* and *user* are injected into the business context to allow reasoning over them. It enables validation of actor's privileges as well as validation of the *product* instance.

Contrary, constants are immutable values configuring the environment. For example, consider VAT rate in a system. This configuration value must be considered in many places throughout the system including business rules. To enable its reuse and avoid manual repetition, it is necessary to support definition of system-wide constants, and make them available and reused in contexts.

Custom functions are an important requirement in case of a universal DSL for business rules. Each domain, or even each system, operates with custom constraints often encapsulated in functions. Furthermore, even same constraint may be implemented differently. For example, consider authentication mechanism. Each system implements a function *`isAuthenticated()`* slightly differently.

There exist two methods to enable this language extension. First, the language is always adjusted to a current domain including a set of functions. This puts high demands on evolution of the language and subsequent tools such as advices and aspect weavers to be able to accept all components[2] of the language. Second, the language supports dynamic functions, which are indirect references to a function dictionary, which may vary per project or even per component. Then aspect weavers accept the dictionary and look up a function. This indirection introduces several issues such as

[2] For example, a function may have different implementation for the domain layer at the server and for the client-side of the UI in the presentation layer. Considering authentication, while the server considers a proper mechanism, the client may use a different mechanism or just return a dummy value.

possibly slower performance, need for multiple implementations a function for different technologies, and lack of static verification that all declared functions are defined in all dictionaries. However, it significantly reduces efforts for continuous evolution of the language and the tools, and reduces the mental barrier introduced by continuously evolving language changing on a project basis.

Platform-independence is an essential requirement for the ADD. While source code of this concern may be platform-specific[3], there must exist a platform-independent representation enabling communication of the rules. Considering the layered architecture, there are various components implemented in different technologies. Moreover, having the implementation of a system distributed over multiple servers, the rules must be transmitted over the network. In consequence, although each component may consider different technologies and be implemented in a different general purpose language, all implementations accept the same representation of the rules, which requires existence of the platform-independent format.

For illustration, consider Java language. Source code is plain text easily understandable to humans. It compiles into bytecode, which is a platform-independent set of instructions and there exist many implementations of the instructions for various environments. Business rules are in a similar situation. Their source code must be easily understandable to domain experts but then must be transformed into some platform-independent format implemented for various aspect weavers.

Context abstraction and composition aim on simplification of maintenance and reduction of duplications. For example, consider user authorization as a subset of business rules. A system recognizes several roles, and execution of each business operation is restricted only to some of them. There exist three approaches to protection in the ADD. First, authorization is implemented directly in a business operation, which is discouraged by the ADD as it should be extracted and separated from business logic. Second, each context of a protected operation defines the restriction rules verifying actor's roles. This leads to high duplication of identical rules across contexts with the same authorization policy. And third, there exists an *abstract* context declaring the authorization policy, which is included in all contexts sharing this policy. This context composition and reuse allows reuse of business rules with positive impact on maintenance efforts as it avoids rules duplication.

> **Definition 5.1** *A business context is abstract, when is not linked to any business operation, and cannot be referenced from application code as it does not directly participate in any use case scenario.*

[3] There may exist a platform-specific editor, e.g., implemented in Java, managing the rules and saving them in a convenient format suitable for the editor.

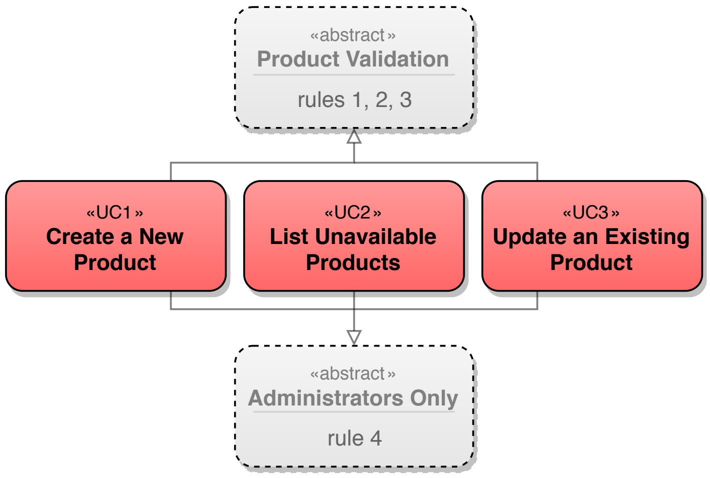


Figure 5.2: Business context composition in the running example

Definition 5.2 (Context Composition) *A business context includes 0..N existing business contexts including constants, attributes, and rules.*

For example, consider the UC 1 from the running example. The operation of product insertion is available only to system administrators (rule ④), i.e., users who were granted role Ⓒ. The business context of this operation then either protects the method itself by declaring `$actor isGrantedOf Role.Administrator`, or includes existing abstract context *Administrators Only*, which implements this rule. Figure 5.2 illustrates the optimized structure of contexts of several UCs from the example. The abstract contexts encapsulate shared rules to avoid their duplication and the non-abstract business contexts relate to actual operations in existing use cases. The impact of context composition on maintenance efforts is obvious. Unfortunately, composition of contexts introduces new level of challenges raised from the order of included contexts, transitive inclusions, cycle detection, and collisions [110].

Inspection of business rules is a new challenge arising from having the rules extracted and encapsulated in the repository. While the separation of concerns simplifies development and maintenance efforts, it puts high demands on the tools. They must inspect the represented concerns, automatically transform them into target platforms, and weave them into existing joinpoints.

Basically, there exist two approaches to inspection of the concern represented in a DSL. The first and common approach relies on parsing of a textual representation of a language into an abstract syntax tree (AST) [110]. Then the AST is transformed into any other form optimal for its further processing. The other method comes from the forward engineering and differentiates a presentation of a language and its representation. While the presentation is what a user sees, the representation describes an internal model [111]. This projection-based approach does not require parsing of plain text as it directly manipulates the AST instead. While benefits of the first approach lie in its simplicity and straightforward use, it requires implementation of a complex parser decomposing relevant language into elements with high-quality error handling. Contrary, the second approach requires implementation of a complex editor and definition of presentation rules but language inspection is very straightforward as it is already represented in a processing-friendly model.

Transformation and distribution are final two requirements defined by the ADD on a DSL for business rules. It is obvious that the rules must be transformed into various forms and languages to be reused throughout the whole system. Considering the layers, it is likely that each of them uses different implementation language for the rules. For example, while the data layer queries a persistent storage using a SQL-like language, the domain layer considers the rules in a general-purpose language used for implementation of this layer, and the presentation layer often contains a client-side component running in a user's browser, which is commonly implemented using a JavaScript. In consequence, transformation of rules and their distribution in a system is essential for successful implementation of the ADD.

Having an AST, the transformation is straightforward. Each target component defines its own mapping rules between AST nodes of the shared and the component-specific trees. For example, some nodes may remain the same, implementation of some predicates can change, and other predicates can be ignored. Then it transforms the shared AST into the component-specific AST, which is finally serialized into a component-specific language. This implementation implicates distribution of the rules in a platform-independent format, which is discussed earlier in this chapter.

In conclusion, design of a novel language for business rules representation for the ADD is a great challenge. Besides the conventional challenges in business rules representation identified and discussed in Section 3.5, the requirements identified in this section put additional demands on design of the language. On the other hand, having this representation enables further reuse of the rules (Objective 2), which is discussed in the next section. Therefore, the efforts devoted to design and maintenance of another language are traded for additional reuse of captured information.

5.2 Use of Business Rules

Having efficient and inspectable representation of business rules makes wide range of their reuse possible. Besides the obvious use within the ADD and their application in architectonic components, it opens new areas significantly benefiting from inspection of business rules. Some applications introduce new aspects and responsibilities into software development, other just deal with issues introduced by the ADD. However, this section briefly suggests new challenges and future research, which are out of the scope of this thesis, but might greatly impact the development process.

Reuse within EISs is already briefly suggested in the previous chapters and is discussed in detail in Chapter 6. While this challenge is here only for completeness, reuse of the rules from the repository in the components of the architecture brings significant benefits, which include major reduction of source code, duplications, and efforts. In consequence, the separation of concerns reduces error-proneness and eases testing.

Formal verification of requirements is a promising area of further research. Consider the functional requirements decomposed and written in a formal language such as the OCL. Then having business contexts decomposed and extracted from a system enables their transformation into the same language. In consequence, it is possible to reason over both contexts and requirements and, e.g., prove that the contexts implement the requirements. Furthermore, using the ADD with other development and engineering methods such as the model-driven engineering, it seems possible to model the requirements in a formal language and then either automatically transform them into the concern or run formal verification to ensure their consistency.

Context feasibility and cycle detection extend the idea of formal verification. The decomposed contexts implementing UCs make new places for human errors. It is easy to make a mistake during manual implementation of the contexts and thus introduction of further testing is necessary in order to assure quality. As is suggested above, transforming the contexts into a formal language enables reasoning over them, and there exist several hypotheses to reason about.

For example, as a business context consists of many rules, it is easy to make a mistake and create an unfeasible context, i.e., define the rules so that there does not exist any combination of inputs evaluating the context to true. Automatic formal verification of context feasibility would detect this kind of an issue.

In order to reuse rules among contexts, the ADD proposes context composition. Unfortunately, having the rules scattered throughout multiple contexts may lead to unfeasible contexts, which is discussed above, but also the context composition may result in cycles in a hierarchy of context. Simple detection of cycles can detect this issue in early stages of development, e.g., in compile time or using a static code analysis.

Validation of requirements by domain experts is the most simple but likely most benefiting feature enabled by inspectable business contexts. Having the contexts extracted and described in a human-friendly domain-specific language, domain experts are able to review the contexts in their raw format or transformed into a summary to validate the operations against the requirements. This validation does not assure quality of implementation, i.e., not do the thing right, but it is a mechanism to assure delivery of what should be delivered, i.e., do the right thing [67].

Business documentation provides an overview of a system, highlights a list of implemented services, their operations, and business rules, which domain experts should

Listing 5.2: Example of a language for business rules description with the ADD

```
abstract context: product validation
  inputs:
    product: Product
  preconditions:
    product.name not empty
    product.price is positive
    product.weight not negative
```

a: Abstract context of product validation

```
context: UC1: Create a new product
  inputs:
    product: Product
  include:
    administrators only
    product validation (product)

```

b: Business context of UC1

validate against the requirements [67]. Unfortunately, although there exists techniques such as a phrasal pattern matching [89], and construction of a call-graph and branching detection [113], extraction of actual documentation is usually very challenging, since manual maintenance of the documentation is tedious and semi-automated reverse engineering techniques require significant efforts and are not accurate [100]. Fortunately, utilization of the concerns maintained in the ADD enables generation of current business documentation using automated forward engineering techniques [A.2, A.6], and thus provides additional value to domain experts.

This list of the challenges is neither complete nor in the scope of this thesis, but it is a subject of future research. Its objective is to provide insight to what could be done with business rules and how their separation and transformation may improve development, especially the aspect of quality assurance. In conclusion, having business rules efficiently separated and ready for automated reuse opens new possibilities and challenges for their further reuse, which balances the efforts required for design and implementation of the representation itself.

5.3 Example of a Language for Business Rules

Considering existing languages discussed in Section 3.5 and the requirements defined by the ADD in Section 5.1, there exists no language addressing them all. In consequence, either an existing language must be customized or a new language must be tailored to be used with the ADD. Unfortunately, design and implementation of a language is a great challenge [72] and thus is left outside the scope of this thesis and saved for a future work.

Nevertheless, for illustration purposes and the proof of the concept implementation of the ADD, a language with a very limited set of instructions but fully addressing all the ADD requirements was designed. Unfortunately, this language is not designed for production use and thus its full specification is not included in this thesis.

For the sake of the following chapters discussing business contexts and illustrating implementation of the ADD in various components of the architecture, consider the example in Listing 5.2. It demonstrates all significant concepts of the language, while it addresses the requirements. The name of the context is in the root of the context followed by a list of input parameters, a set of included contexts, a set of preconditions, and a set of postconditions. For the purpose of this thesis, the language is implemented in JetBrains MPS[4], which is a tool with a projectional editor easing implementation and transformation of DSLs. Easy transformation provided by the MPS allows producing a multiplatform representation of the contexts ready for further transformation, and thus addresses the most significant requirements. In conclusion, although the example presented in this section is not ready for production use, it is used in the examples in Chapters 6 and 7 to demonstrate the ADD and captured business rules.

5.4 Summary

The previous chapter introduced the aspect-driven development approach addressing the separation of concerns in enterprise information systems (Objective 1). The approach recognizes the importance of business rules in a system, considers them among other concerns, and provides a mechanism for their separation and subsequent reuse (Objective 2). Although, the complexity and the manner of utilization of business rules in the ADD defines new requirements additionally constraining their representation. Therefore, in order to fully address Objective 2, this chapter elaborated the requirements and design of business rules representation. Unfortunately, since there exists no language or representation already addressing all these requirements, and tailoring a new language is a major challenge requiring significant amount of work, design of such a new language is not in the scope of this thesis and is left for future work. As a result, this chapter further elaborated utilization of rules represented in an inspectable form, which balances the efforts required for initial design and development of the language.

Fortunately, the ADD does not enforce any particular implementation of a DSL, and it is agnostic to any implementation details. There exist major benefits of the ADD being agnostic to a language. First, the language can be customized to a particular domain of an EIS. While it introduces significant work in subsequent tools, the resulting efficiency might outweigh it. Second, the language can continuously evolve and introduce new features such as support of error messages when a rule is not satisfied.

In consequence, while design of the DSL is not significant for this work, for illustration purposes, Section 5.3 briefly introduces an example of a novel language addressing the requirements. The language is used in demos implementing the ADD later in this work. Nevertheless, this incomplete language is neither developed with domain experts nor tested, and thus it is not intended for production use or to be published with this thesis. Design of such a production-ready language is left for future work.

[4] Available online at https://www.jetbrains.com/mps/.

Chapter 6

Implementation of the Approach

The aspect-driven development approach focuses on the separation of concerns in EISs via utilization of multiple existing approaches. It modifies an existing architecture, while it remains agnostic to any implementation details. This chapter presents several implementations of the approach into design of common components of the layered architecture[1], and discusses its utilization in a distributed environment of the service-oriented architecture to address Objective 3 of this thesis. The examples are implemented via formalization and mapping of the approach into the terms of the AOP, which is the essential underlying approach. The examples suggest intended use of the ADD and allow its further evaluation, which is discussed in the next chapter. However, this chapter does neither show nor suggest any technologies or source code fragments, as the development approach operates on a higher level of abstraction and affects design instead. Finally, the examples are not the only weak spots in EISs, there exist more components potentially benefiting from the ADD utilization.

Consider an incoming request into a system fired by a user[2]. The flow of the request throughout the system is suggested in Figure 6.1, which highlights all points considering cross-cutting concerns and thus suffering from manual information repetition, concern tangling, and high error-proneness. First, when a user fires an action in the UI, an event is emitted and either handled by the UI or directly or eventually[3] submitted to the server. The server-side of the UI receives the request and validates the input to avoid inconsistencies or hijacking of the UI. Then, the request is processed and the response is served, which may involve invocation of the domain layer. Operations in this layer are invoked from many places so it is easy to have some inconsistencies in context validation procedures. In order to guarantee assumptions of an operation, each business operation implements a checkpoint, which revalidates the execution context

[1] Since the ADD is agnostic to implementation details, this thesis demonstrates it on the conventional layered architecture [28], which usually consists of the presentation, domain, and data layers [42].

[2] In case of B2B communication the UI is replaced by some API, otherwise it is similar.

[3] While a rich/thick client-side application submits data to server only if it is valid, a thin client-side application submits data directly to the server without further delay.

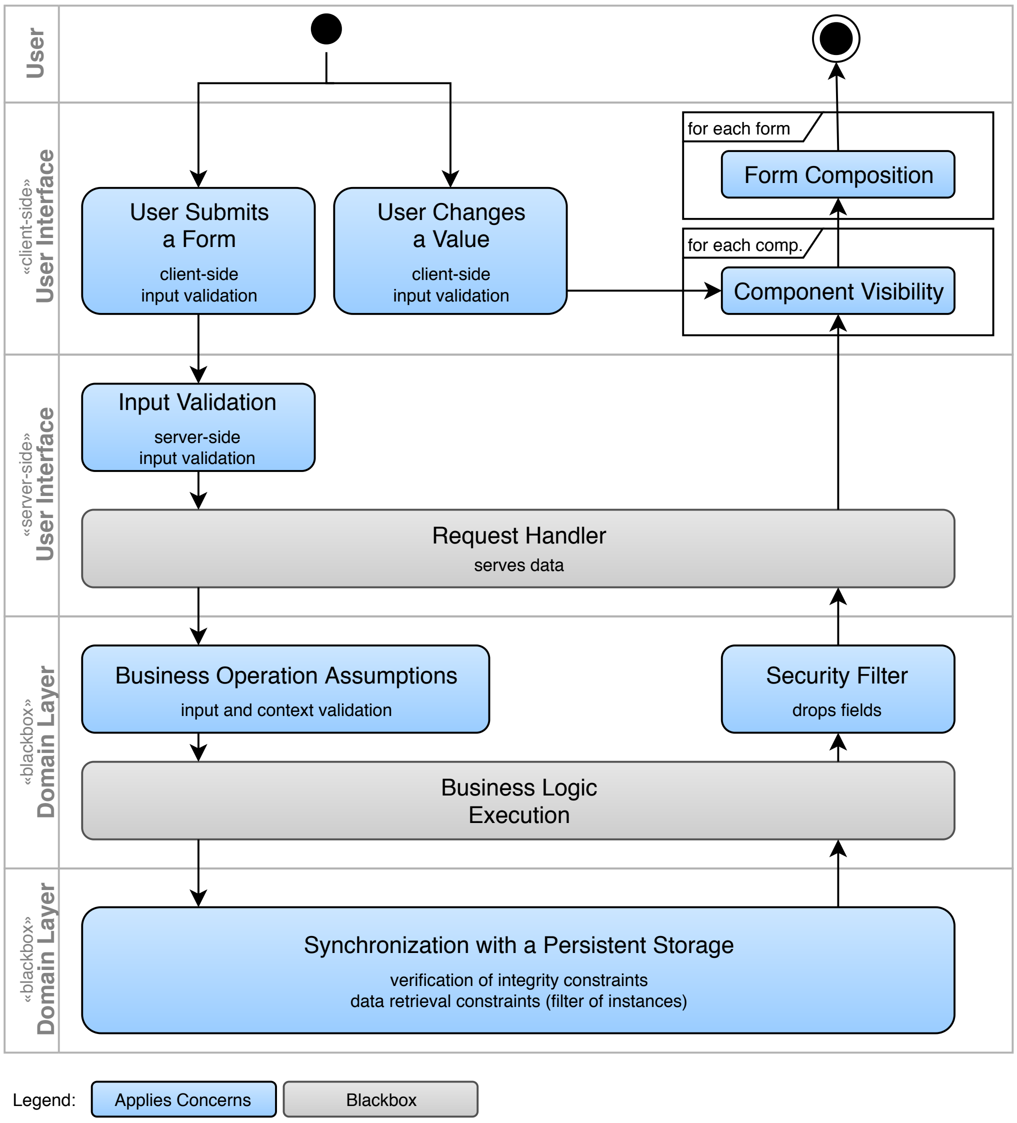

User
«client-side» User Interface
«server-side» User Interface
«blackbox» Domain Layer
«blackbox» Domain Layer
User Submits a Form
client-side input validation
User Changes a Value
client-side input validation
for each form
Form Composition
for each comp.
Component Visibility
Input Validation
server-side input validation
Request Handler
serves data
Business Operation Assumptions
input and context validation
Security Filter
drops fields
Business Logic Execution
Synchronization with a Persistent Storage
verification of integrity constraints
data retrieval constraints (filter of instances)
Legend:
Applies Concerns
Blackbox

Figure 6.1: Request flow throughout an EIS and the life-cycle of the UI

including the input data. A business operation then usually queries a database or some other persistent storage. Each query has to be context-aware, i.e., consider a current execution context, to restrict a view to data and ensure that the user and the operation are eligible to see it and operate with it so it often includes checking of permissions and validity flags. When the operation exits, the output data are filtered to drop all instances and erase all attributes the user is not eligible to see. Finally, the response is eventually[4] sent to the client and the the UI is constructed or updated.

While the invocation flow of a request displayed in Figure 6.1 is straightforward, many points consider various cross-cutting concerns (Problem 1), which tend to cause problems highlighted in Section 1.2. Furthermore, these points are distributed throughout a whole system, i.e., throughout multiple layers and technologies (Problem 2),

[4] While a rich/thick client-side application constructs forms and other components of the UI itself, a thin client-side application already receives these elements constructed and only renders them.

Table 6.1: Utilization of the ADD in the layered architecture

Sec.	Intention	Concerns/Aspects	Use of the ADD / Weaving
6.1	composition of the UI	data model, business rules, localization, layouts, widgets	value change, form submission, form composition, conditional rendering, input validation
6.2	composition of the distributed multi-platform UI	data model, business rules, localization, platform-specific widgets, layouts	value change, form submission, form composition, conditional rendering, input validation
6.3	input validation in the domain layer	data model, business rules	protection of business logic as input validation, security filter
6.4	querying of a persistent storage	data model, business rules	storage querying and output restriction
6.5	reuse of concerns in the SOA	data model, business rules	inter-service reuse of business rules and a model structure

which significantly impacts identification, representation, and reuse of the concerns (Problems 3 and 5). As such, this chapter elaborates all these points and proposes an implementation of the ADD into design of impacted components to efficiently maintain the concerns and address the problems. In addition, Table 6.1 summarizes implementations the ADD discussed in this chapter, their focus, and considered concerns, which arise from the points in the flow. Therefore, Section 6.1 elaborates implementation of the approach into design of a UI component of the presentation layer, and Section 6.2 adjusts the design for a distributed multi-platform UI. Section 6.3 discusses design of the domain layer to reuse the concerns to validate the input of business operations, and Section 6.4 proposes modification of the design of the data layer to automate construction of queries into a persistent storage. Finally, Section 6.5 scales the approach into the scope of the service-oriented architecture to enable cross-service concerns reuse.

6.1 Presentation Layer: User Interface

The UI is an important but complex component of most EISs. It exposes a system to users and lets them manipulate data through defined scenarios. The UI component consists of a server-side and a client-side as is illustrated in Figure 1.1. The server-side accepts incoming requests, validates input data, and serves responses. Assignment of responsibilities for UI composition and user experience (e.g., client-side validation) varies based on implementation of the UI [18]. However, the UI composition and input validation are complex tasks considering many cross-cutting concerns as defined in Section 3.4.3. Since the concerns are tangled together, source code of the UI tends to suffer from problems discussed in Chapter 1. In consequence, this section demonstrates simplification of the UI development through utilization of the ADD and the separation of concerns. For illustration purposes, this section assumes a rich client implementing client-side data validation as a thin client is only its simplification.

There exist many concerns in the UI, however, a data model, UI widgets, UI layouts, localization, and business rules are the most significant, which is discussed in Section 3.4.3 and Section 4.1. Consider Figure 6.1, which also displays a life-cycle of the UI, although it is simplified for the purpose of this chapter. The diagram highlights five points[5] in the UI, each restating a subset of these concerns. Unfortunately, each of them applies the concerns differently, which is elaborated in the following text.

First, when a user changes a value of an input and there exists input data to be processed, the client-side of the UI validates them against an execution context. It applies a data model and business rules matching the context and evaluates them with given parameters. If the rules are satisfied, the invocation of the action continues and the UI is updated, otherwise the UI displays an error considering a localization concern.

Eventually, each form is supposed to be submitted or some other action is posted to the server. Then the input is validated using a data model, business rules, and an execution context, and then submitted to the server if all rules are satisfied. In case of an error, a user is notified and the UI is updated.

When the server-side of the UI receives a request, it validates input data and an execution context. This double check solves any inconsistencies between the server-side and the client-side plus prevents the UI hijacking. Furthermore, server-side validation may involve additional rules, which cannot be checked at the client-side for either security or performance reasons. If all rules are satisfied, the response is preprepared. It usually involves invocation of a business operation in the domain layer, which is a black box in this scenario. In case of errors, the client is notified.

While validation at the client-side must produce user-friendly errors, the server-side validation only aggregates them and passes them into the client-side for further processing such as localization. However, both components must eventually produce user-friendly errors as they likely occur. Contrary, the domain layer may assume only valid inputs so it may fail with less user-friendly errors, if the assumptions are violated because it indicates a bug or invalid use rather than user's error.

When request handling ends, the UI must be updated at either the server-side or the client-side depending on implementation of the UI. It includes a) showing/hiding conditionally rendered components, b) composition of input forms from their data models, and c) final composition of UI components. While the last step is straightforward and well implemented in many templating frameworks, the two other steps involve several concerns and require their additional repetition.

[5] These points are joinpoints in terms of the AOP, which is discussed later in this section.

A conditional component is rendered only when a condition is met. This condition is usually a precondition of a business context of a related business operation, so it considers a data model, a current execution context, and a business context of that operation. While implementation of such a conditional component is easy, it quietly pollutes source code with repeated concerns. Furthermore, due to high scattering of these conditions, they tend to get obsolete during system maintenance.

While some concerns are significantly scattered and repeated in the previous examples, they all tangle together in input forms in the UI [59, A.3]. The forms (possibly in a read-only mode) allow users to browse and fill data hence they are essential in a system. A form consists of inputs mapped into the fields of a data model[6], presented via widgets, and organized into a layout. Every field and the model itself is constrained by business rules depending on the execution context, i.e., set of rules attached to a current business operation considering a user context, an application context, and a request context. Finally, a label of each field is extracted from the model and localized.

In conclusion, there exist significant concerns repetition and tangling in the UI. While localization, widgets, and layouts repeat only horizontally in the UI, i.e., across multiple forms, business rules and a data model repeat both horizontally and vertically, i.e., also in both sides of the UI and even outside the UI. As the concerns are significantly scattered and tangled together, their linearization results in complex source code with difficult and error-prone maintenance [A.3].

6.1.1 Utilization of the ADD

The aspect-driven development approach is focused on the separation of concerns, which has major potential in the UI as it is where many concerns are tangled together [17]. It recognizes these concerns as independent disjunct aspects and describes them separately in convenient DSLs, which is suggested in Chapter 4. The decomposed concerns are located in the repository as is hinted in Figure 4.2, and then weaved back together at runtime by multiple aspect weavers specific to a context of use. The weavers differ in their output as well as the injection points differ in their position in the flow. Context-awareness allowing system customization for each invocation is the major advantage of runtime weaving. For example, system may behave differently for users from the United States, e.g., it requires also a state when filling an address.

The AOP, which is significantly utilized by the ADD, recognizes several essential elements (aspects, joinpoints, pointcuts, and advices) participating in the separation of concerns and subsequent weaving. In order to apply the approach, the mapping between the UI and the elements of the AOP must be defined to establish understanding of the domain (the UI in this case) and hint implementation of the ADD.

[6] The domain layer, a server-side, and a client-side of the UI usually operate with different models.

An aspect in the UI is a concern in the UI, which tends to tangling and scattering. This thesis recognizes the following concerns as aspects in terms of the AOP:

(i) *A data model*, which is behind an input form,
(ii) *UI layouts*, which define organization of input fields into forms,
(iii) *UI widgets* presenting model fields as input components in the UI,
(iv) *localization* defining translations for labels and error messages,
(v) and *business rules*, which define constraints over form fields and a form itself.

Although this set of concerns is project-specific, the concerns are tangled and scattered throughout the UI, which leads to major code repetition. The ability to extract them and organize in the repository significantly eases their readability and maintenance. In addition to other benefits mentioned in Chapter 4, the concerns are described in DSLs.

Advice in the UI is functionality to be weaved in varying for each concern, as each concern represents a different domain. Considering the concerns, some of them cross-cut throughout all layers of a system, while other are specific to the UI. In consequence, while the latter concerns can be in a UI-specific form, the former concerns are stored in the repository in a generic form and must be further transformed before they can be weaved into the UI. The advices of the concerns are implemented as follows.

(a) As *the model* cross-cuts through both server-side and client-side of the UI, the advice must be in a generic form acceptable by both sides. In consequence, the model is inspected and serialized into a general format including a list of fields and their names and types, and then passed into an aspect weaver.
(b) A *layout* advice is an integration template with placeholders [17], into which the fields are injected. Concrete placeholders explicitly determine the order; implicit placeholders consider weighted or alphabetical order of model fields.
(c) A *widget* advice is a presentation template for a single field, possibly matching only certain conditions. For example, a widget for a simple string is different than a widget for a password, which is also a string. A widget template is described in a DSL specific to a technology of the UI.
(d) A *localization* advice is a single record from a localization dictionary. The record is a tuple of a unique convention-driven key and a textual value.
(e) Finally, advice of *business rules* is an implementation of verification of a single condition from a related business context, i.e., a set of business rules [A.13]. First, the condition is transformed into runnable code specific to an invocation platform (vertical transformation), and then passed into an aspect weaver for further composition with other concerns (horizontal transformation).

Joinpoints in the UI are injection points in the UI's life-cycle (Figure 6.1), ready for the concerns to be weaved in. The flow highlights five injection points:

① A user interacts with a system by changing values of input fields. *Change of a value* is an action performed by a user, when an input in a form is changed.

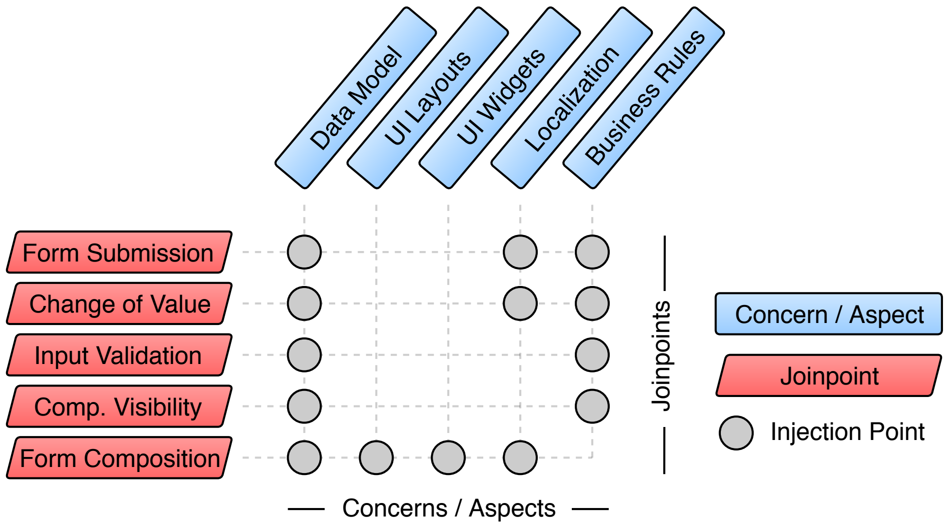


Figure 6.2: Pointcuts of concerns in the UI

② Each form is intended to be eventually submitted. *Form submission* joinpoint is intended for submission of data to a server for its further processing.

③ Incoming data must be validated at the server to prevent UI hijacking and verify assumptions. *Input validation* at the server-side verifies assumptions of an action.

④ During the UI composition, conditional components are shown/hidden according to their conditions. *Component visibility* joinpoint is a step in the UI composition process responsible for showing and hiding conditional components.

⑤ During the UI composition, input forms are constructed. Each form is a presentation of a data model composed of multiple concerns. *Form composition* joinpoint is a step in the UI composition process considering all UI concerns and composing them into a form backed by a data model.

Pointcuts in the UI select a subset of UI joinpoints to apply a particular advice, which is denoted in Figure 6.2. The pointcuts accept an execution context providing runtime parameters to include dynamic joinpoints.

However, as an execution context is a complex structure encapsulating all existing subcontexts as defined in Chapter 4, only some of them are actually considered by each pointcut. Table 6.2 summarizes tuples of a concern, a pointcut, and contexts to indicate, which contexts and concerns are considered by which joinpoint (a pointcut selects multiple joinpoints). A data model is applied in all joinpoints, and the advice is derived from a business context, e.g., which model class is being presented. UI Layouts are applied in ⑤, an advice is selected according to the request context, which provides runtime parameters such as user's device and screen size. UI Widgets are also applied only in ⑤, but an advice selection is a complex expression considering all runtime parameters in an execution context and also a type of a field. For example, a business context determines if a field is read-only according to current timestamp and

Table 6.2: Context-aware pointcuts in the UI

Concern	Pointcut	Considered contexts
Data Model	① ② ③ ④ ⑤	Ⓑ
UI Layouts	⑤	Ⓡ
UI Widgets	⑤	Ⓔ = Ⓐ Ⓑ Ⓡ Ⓤ
Localization	① ② ⑤	Ⓡ Ⓤ
Business Rules	① ② ③ ④	Ⓔ = Ⓐ Ⓑ Ⓡ Ⓤ

Ⓐ Application context Ⓔ Execution context Ⓤ User context
Ⓑ Business context Ⓡ Request context

geographical zone. Localization pointcut selects an advice for dynamic joinpoint ①, ②, and ⑤ based on request and user contexts, i.e., it considers user's language preferences. Business rules are applied in ①, ②, ③, and ④ and consider a full execution context to support restrictions over all contextual variables [A.13]. In conclusion, although an aspect weaver expects a whole execution context, its components are only used selectively.

The important benefit of the ADD lies in the context-awareness and dynamic joinpoints. Although there exist approaches addressing the separation of concerns and automated composition, they do not recognize dynamic joinpoints, which support runtime parameters as an execution context. To illustrate the benefit, consider different widgets for desktops and mobile devices with much smaller screen. A pointcut is resolved at runtime and therefore an alternative responsive variant of a web page does not have to be prepared. A dynamic selection of joinpoints results in a more compact layout and responsive variants of UI widgets instead. Furthermore, some fields may be added or hidden according to the user's geographical location or a timestamp.

Aspect weaving in the UI is runtime application of advices and construction of output, which is either source code or directly executed behavior. An aspect weaver accepts a set of aspects and each weaving invocation is parametrized by the execution context, which includes a position in the execution flow or source code. The weaver selects joinpoints using pointcuts, vertically transforms advices if needed, and combines them into output, which differs per position in the UI life-cycle and intended use.

To illustrate the weaving in the process captured in Figure 6.1, consider a user interacting with the UI of a system. First, a user changes a value of an input in a form, which triggers a change event. The event handler invokes an aspect weaver instructing it with the field, the data model of the form, and the execution context. The weaver extracts applicable business rules from the execution context and applies them to the field. In case of validation errors, the weaver applies localization advices to localize the errors and returns the validation results to the handler, which resumes invocation. Note that each form relates to a business context, which is used for input validation.

Eventually, a user submits a form or posts some other action, which triggers a submission event. The event handler invokes an aspect weaver instructing it by a data model of a form and an execution context. The weaver validates the model with business rules in the execution context and returns the validation result to the handler, which resumes the event handling.

The server-side of the UI exposes multiple action handlers to be invoked by a form submission or other action, each of them representing a business operation. In consequence, a handler relates to a business context defining assumptions of that operation, i.e., business rules constraining input data and a context. When an action handler is invoked, an aspect weaver intercepts the invocation and validates the input and an execution context with the related business context. This validation may also control security preconditions to determine whether a user is eligible for the operation and for reading/changing each field. When the validation is finished, the weaver passes the validation results to the handler and resumes its invocation. Then, the handler usually invokes some business operations in the domain layer, but this part of the process is discussed later in this chapter. For the purpose of this section consider the domain layer to be a blackbox. Finally, the handler serves a response to the client. In consequence, for each action handler, developers define a reference to an existing business context to be used for context-aware input validation.

In the end, each event handler updates the UI. First, conditional components, i.e., components to be rendered only when a condition is met, are reevaluated to show/hide them considering the execution context. The condition in these components is usually a precondition to some operation, i.e., it is a business context of that operation. In consequence, each conditional component references an existing business context. Then the aspect weaver intercepts the UI composition to evaluate the business context using parameters in the current execution context and either shows or hides the component.

Finally, the UI of EISs consists of many input forms. An input form is a presentation of its data model considering several other concerns. In consequence, the concerns of an input form are decomposed using the ADD, and then each form is automatically constructed using the aspect weaver. An input form references its data model defining a list of fields, their names and types, a business context with validation rules. Then, there exist UI layouts and UI widgets with mappings defining their use. To construct an input form, first, the layout advice is applied, interpreted, and processed into an integration template [17]. Second, the weaver iterates over fields in the model, selects the most appropriate widgets, localizes the keys, and weaves them into the layout. As a result, the output of the weaver is source code of the form, which is then rendered and prepared for interaction.

In conclusion, while decomposition of the concerns is straightforward and intuitive, implementation of aspect weavers remains challenging. Dynamic joinpoints, parameters in pointcuts, and diversity of DSLs significantly increase complexity of weavers. Considering the request life-cycle, multiple different weavers accepting same aspects but producing different output must exist. Finally, while the separation of concerns is manageable, the initial overhead and complexity of implementation of the weavers introduces a major barrier. On the other hand, choosing efficient technologies and having generic implementation of the weavers results in project-agnostic weavers reusable across multiple projects, which may significantly utilize the initial costs.

6.1.2 Summary

Since an EIS implements various business processes consisting of business operations, its UI is rich and complex as it consists of many input forms and exposes a lot of actions. Implementation, maintenance, and keeping all models and business rules constraining the operations across all forms, client-side validators, and action handlers consistent is challenging and error-prone as conventional approaches and technologies tend to significant information repetition and scattering. Moreover, maintaining concerns in multiple languages and technologies (one for server-side, other for client-side, and another for the data layer) increases both expenses and mental barrier.

The separation of concerns in EISs implemented by the ADD significantly reduces the complexity of the UI. The tangled concerns can be decomposed and described independently. Then, the aspect weavers reuse the concerns already used by lower layers of a system and combine them with the UI-specific concerns to deliver context-aware UI without manual information repetition. This thesis recognizes five points in the life-cycle of the UI considering the concerns and thus proposes utilization of five aspect weavers, each tailored for a single point. The context-aware weavers weave concerns together at runtime, which automatically transforms and distributes the concerns, and subsequently saves efforts and removes error-proneness.

There exist two levels of the context-awareness in the UI with the ADD. First, declared business contexts accept any contextual information provided to a weaver [A.8], e.g., user's identity and security roles, and consider it in business rules. Second, the pointcuts consider the execution context including the target location, which enables advanced runtime selection of advices. For example, it picks a different UI widget for a small device or a different widget if a textual field is marked as a password.

Finally, although the ADD significantly reduces complexity of UI maintenance, it introduces significant initial overhead and both mental and technological barriers. To fully implement the ADD in the UI, there must exist DSLs for each concern and all five aspect weavers automatically reusing the concerns. Doing so for a single project would be inefficient. Fortunately, neither the weavers nor DSLs depend on a project, they depend only on technologies, so they are reusable across multiple projects, which utilizes the initial efforts and increases the efficiency of the approach.

6.2 Presentation Layer: Distributed User Interface

Enterprise information systems expose their interface to users to let them browse and manipulate data through various business processes. Unfortunately, the UI consists of significant amount of input forms and exposes many operations, which results in tangling of multiple cross-cutting concerns. While the previous section utilizes the ADD in the UI to reduce efforts and complexity of its development and maintenance, contemporary systems tend to implement a distributed UI, which comes with additional challenges. While conventional systems usually support web browsers, modern applications deliver native applications for mobile devices to increase user experience and utilize potential of these platforms. Unfortunately, although provided functionality is identical, there are limited possibilities to reuse concerns across mobile platforms and web browsers due to diversity of technologies and programming languages. This section introduces a modified version of the aspect-driven development to support a distributed user interface and enable cross-platform reuse of cross-cutting concerns.

Contemporary EISs and web applications often use a rich client application to implement the UI, i.e., a client-side standalone application backed by a server providing data via services. These client-side applications of the presentation layer usually run in web browsers, and thus significantly increase user experience, reduce response time, and deliver experience similar to desktop applications. Considering expansion of mobile devices, many applications and EISs also target mobile devices besides web browsers at desktops. However, as mobile devices have specific screen size as well as lower performance, the systems often come with native mobile applications tailored for a particular mobile platform [115]. It enables additional customization, use of specific aspects of mobile devices and improves the user experience. Unfortunately, although implementations for web browsers and all mobile devices are essentially identical, diversity of platforms[7] significantly increases development efforts as there exist limited possibilities to reuse concerns and models across platforms.

Consider the request flow and the life-cycle of the UI suggested in Figure 6.1. Standalone client applications do not reload between requests, they communicate in background and update the UI according to received responses. However, although the architecture of client-side applications is different comparing to thin clients and they live much longer without reload, the essential communication schema is still valid including all points considering the concerns. The client-side applications consider

[7] Nowadays, there exist standardized web browsers and three major platforms for mobile devices: Android, iOS, and Windows Mobile.

three platforms instead of one, nevertheless. While simple EISs provide only implementation for web browsers, contemporary systems often focus on mobile devices, and besides a web application deliver also an application for Android and an application for iOS. In consequence, there are three implementations of the UI instead of one, and subsequently, there are also three-times more points considering the concerns in the client-side of the UI. For example, consider the composition of input forms. This complex task assumes at least four significantly repeated and tangled concerns, as is discussed in the previous section. In case of three different platforms, the concerns are transformed into three technologies and the form composition is implemented for each platform. Therefore, although each application follows same specification, limited reuse of concerns significantly impacts development and maintenance efforts.

6.2.1 Utilization of the ADD

The the mapping of the UI into the AOP defined in the previous section enables reuse the concerns located in the concerns repository, and implementation of several aspect weavers to make the UI dynamically composed at runtime without any need for manual concerns restatement. Unfortunately, this mapping does not address the distributed UI, since some concerns such as a data model and business rules are platform-independent and get distributed, while other concerns such as UI widgets are platform-specific. Consider a platform-independent concern, e.g., business rules. The concern, in terms of the AOP is an aspect consisting of advices. To have the concern separated and to be able to weave it automatically back into source code, the pointcuts must be able to identify, where to weave each advice. To do so, an advice has its own address, which is the name of the business context in case of business rules. Then, this address is used instead of manual repetition of advices.

> **Definition 6.1** *A platform-independent concern consists of advices, which are transformed via forward transformation rules into various target platforms. Each advice has its address to be used as a marker for aspect weaving within source code.*

However, this address is not a sufficient identifier for platform-specific concerns. For example, a UI widget for password is different for web browser and for Android. As a solution, there exists a composite key consisting of an address and a platform.

> **Definition 6.2** *A platform-specific concern consists of advices implementing the concern for a single platform. Each advice has a composite key consisting of an address in the repository and a platform.*

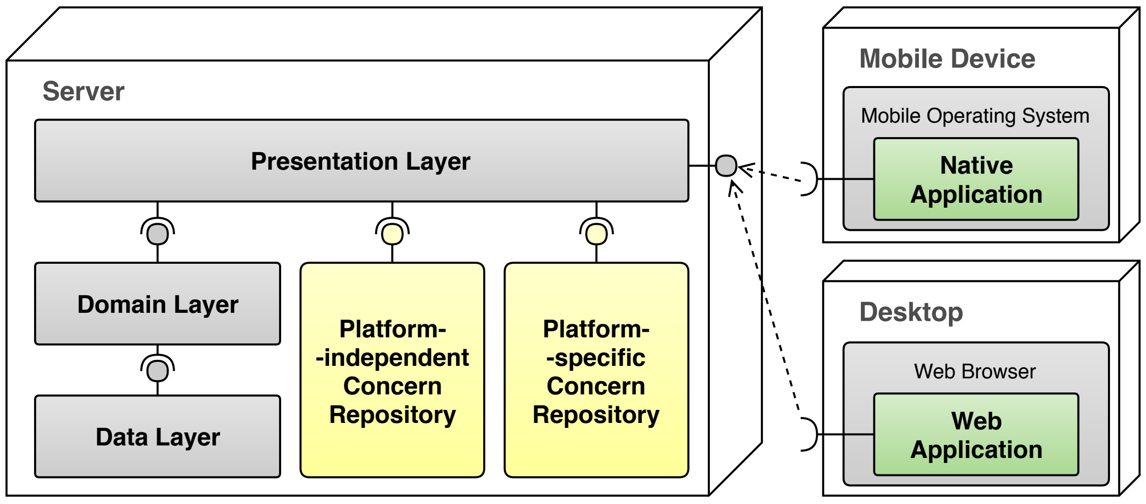


Figure 6.3: The layered architecture for the distributed UI

Considering the differences in platform-independent and platform-specific concerns, there exists modified implementation of the ADD for distributed UIs [A.1]. While the mapping into terms of the AOP, and identification and definition of joinpoints remain same, the architecture of the system is modified. Chapter 4 defines the repository as a component parallel to any existing architecture and introducing a storage for all separated concerns. The implementation of the ADD for the distributed multiplatform UI divides the repository to two parts: a) platform-independent concerns, and b) platform-specific concerns, which is illustrated in Figure 6.3. While the repository for platform-independent concerns follows the specification from Chapter 4, the repository for platform-specific concerns maintains same concerns for various platforms. For example, a data model, business rules, and localization are reused throughout a whole system and thus are maintained in the platform-independent repository. Contrary, UI widgets are platform-specific, each technology implements them differently, thus the platform-specific repository maintains these concerns for each platform.

Having platform-specific concerns maintained separately requires additional validation to avoid inconsistencies. It is essential that for each defined address and all platforms there exist an advice to avoid potential issues and undefined states. Consider an input field for a password. While it might be defined for Android, it would cause a fatal error if its implementation for web browsers was missing.

When the concerns are separated and defined for all platforms, then each native application only performs aspect weaving and does not contain any concerns itself. They are all fetched from the repository in background. In consequence, since the aspect weavers are technology-specific but reusable across projects, there might exist a generic native application interpreting and weaving the concerns fetched from a server and presenting them to users. Essentially, it would be like a web browser for custom languages. However, while this challenge will be addressed in a future work, the implementation of the weavers delivering the UI without any information packed inside the application is covered in this thesis [A.1].

The benefits of the separation of concerns in a distributed UI are significant and include reduced development and maintenance efforts, involvement of domain experts, and reduction of error-proneness. Consider a scenario, where three native clients exist and the application is continuously evolved. As a result, there are many versions of the application for each platform. The concerns are not only scattered but also versioned. Having the concerns separated and maintained in the repository significantly reduces evolution efforts of native applications, since they deliver a version of a concern provided by a server. On the other hand, having multiple platforms requires multiple implementations of the aspect weavers, which introduces significant initial barrier and efforts. Although with all DSLs and the repository already prepared, the benefits of the weavers are significant comparing to the efforts of implementation of the weavers.

6.2.2 Summary

Contemporary EISs often aim on mobile devices due to better user experience. Beside web browsers, there exist multiple mobile platforms, which results in multiple implementations of a client-side application. Unfortunately, the possibilities of cross-platform source code reuse are limited, so developers repeat the concerns manually in source code, which is expensive, error-prone, and tedious, especially when the implementations follow same specification.

The ADD focused on the separation of concerns in EISs saves maintenance efforts, reduces error-proneness, and enables domain experts to be involved in development. However, the original proposal of the ADD from Chapter 4 does not efficiently address the issues of multi-platform applications in the distributed environment. The modification of the approach presented in this section addresses this issue through an extension of the architecture. The novel platform-specific concerns repository maintains platform-specific concerns for various platforms, while platform-independent concerns remain shared and reused within the platform-independent repository. This extension of the approach preserves the separation of concerns, context-aware runtime weaving, and involvement of domain experts into development, which are the most significant benefits of the ADD. Unfortunately, design of the approach and utilization of the AOP introduce major initial barrier and overhead. They require designing DSLs and implementing of aspect weavers to be able to fully benefit from the ADD. On the other hand, the weavers as well as DSLs are project-independent and therefore reusable across multiple projects, which means the initial efforts can be divided among several projects.

In conclusion, future work suggests a possibility of project-independent native applications, which do not contain any project-specific information. These server-driven applications work like web browsers but for custom DSLs, and fetch all concerns and configuration from a server. Their existence would remove the need for manual development and maintenance of native applications, while they would still exist. However, the gap between the solution proposed in this section and fully independent application is still an open challenge, which lies especially in the UI, which does not consist only of buttons and input forms, but also of other elements.

6.3 Domain Layer: Input Validation

The domain layer of an EIS implements business processes of a business domain, which is defined in Chapter 2. Each process consists of steps implemented as business operations in this layer, and as an operation takes a precise position within a process, it defines its assumptions (preconditions) and outputs (postconditions), which create a business context as it is defined in Chapter 4. As the operations directly manipulate data and expose them to users, it is essential to verify the assumptions before execution of an operation to protect business policy and data consistency. However, while many business rules (i.e., both preconditions and postconditions) repeat across multiple business operations, and their business contexts share common ancestors, there is a tendency towards significant manual repetition of business rules and subsequent difficult and error-prone maintenance (Problems 1 and 5). In order address these problems, this section utilizes the aspect-driven development approach in design of the domain layer to reduce the repetitions and maintenance efforts.

Following the request flow in Figure 6.1, upper layers in the architecture invoke operations in the domain layer from various locations. It is obvious that it is difficult to verify the assumptions before invocation of an operation, as the rules would be significantly more restated, which would potentially lead to frequent inconsistencies across locations. In consequence, each business operation must verify its own assumptions itself before it actually executes business logic. However, considering relationships between business contexts (Definition 4.1), there are still significant repetitions across operations. Unfortunately, although there exist approaches to efficient business rules maintenance as is discussed in Section 3.5, they are limited in transformation and reuse of the rules outside business operations, which is an essential requirement for separation of concerns and their automated runtime transformation and weaving in EISs.

For example, consider the *UC 1: Create a new product* from the running example. Although this operation is invoked only during product creation and thus there exists a single source of invocation, the validation rules applied by this operation are shared with the operation updating an existing product in *UC 3: Update an existing product*, which is illustrated in Figure 5.2. Although Section 3.5 discusses methods to reuse the rules across operations, these often fail in their extraction and subsequent transformation into the UI, to deliver the context-aware UI, as is proposed in Section 6.1.

6.3.1 Utilization of the ADD

Since the aspect-driven development addresses the separation of concerns, and introduces the concerns repository to maintain them, the implementation of the approach into the domain layer is straightforward. Having the contexts described in a DSL, each business operation addresses its context in the repository, e.g., by a name. Finally, an aspect weaver intercepts invocation of each business operation to fetch its context, verify the assumptions, and either resume invocation of the operation or throw an exception. However, it is necessary to define a formal mapping between the concerns in the domain layer and the AOP, which is an important mechanism of the ADD.

An aspect in the domain layer is a concern within this layer, which tends to manual restatement, scattering, and tangling. This thesis recognizes business rules to be a significant concern of an entire EIS, as they define business processes, which fulfill the purpose of the system. In the domain layer, the rules validate input data and an execution context, e.g., user's roles, a current timestamp, and a geographical location. Considering the domain layer, business rules are easy to separate and describe in a DSL as well as to subsequently execute to validate data, which is already discussed in Section 3.5. However, they are located in an independent DSL in the repository to fully implement the ADD and enable their reuse also in other components of a system. Besides the benefits of the separation of the concerns discussed in Chapter 1, it brings reduction of source code complexity, because a business operation no longer validates its input itself, but assumes that the input is valid instead. Furthermore, having business contexts described separately in a domain expert-friendly DSL enables easier extraction of current configuration and validation by domain experts.

Advice in the domain layer is business rules, which are weaved via business contexts. The contexts are divided into abstract and non-abstract as is defined in Chapter 5, and the non-abstract contexts are transformed via a vertical transformation into an executable form. Then, they are weaved into joinpoints, which is discussed later.

While the assumptions are valid if and only if they are all satisfied, depending on implementation, there might be more fine-grained identification of rules to provide better validation result in case of an error. However, these assumptions are not supposed to be invalid. This validation only exists to protect data and business policy, and it is not intended to provide user-friendly errors. That is responsibility of the presentation layer, which conducts contextual validation and provides user-friendly errors.

Joinpoints in the domain layer are two injection points in the request flow (Figure 6.1), which accept the weaved concerns:

① *An entrance of a business operation*, is the part of the execution flow, when the operation is called but not yet executed. It is when the business context is to be verified to protect data and a business policy. Verification happens before execution of business logic in order to prevent it in case of violated assumptions.

② *An exit of a business operation* is a point in an execution flow, when the operation execution is finished but before the flow continues in the caller. A security filter at this point optionally erases some fields according to postconditions, if the user is not eligible to see them. However, this use of the concern requires support in the DSL for business rules, since it must contain the information.

Pointcuts in the domain layer select a subset of joinpoints for each aspect applying a particular advice. Considering the domain layer and verification of business contexts, each business operation has to reference its context. Then a pointcut matches the reference to a context to be replaced with an advice, i.e., executable business rules.

Aspect weaving in the domain layer accepts a business operation and a business context, and produces a wrapped operation performing input validation and output filtering. The wrapper accepts an execution context to bind parameters in business rules and then evaluates them. In case of violation of business rules, it throws an exception, otherwise it calls the nested operation with the original input, but then the operation can rely on satisfied assumptions. Similarly, the wrapper intercepts the return values to clear fields of objects, if the DSL provides this information.

In conclusion, implementation of the aspect-driven development into the domain layer is straightforward. The aspect weaver intercepts incoming calls to a business operation and replaces it by a wrapper. The wrapper validates input data and an execution context, and filters returned objects considering a related business context.

6.3.2 Summary

This thesis identifies business rules among significant concerns of EISs because they control business processes, which fulfill the purpose of a system. Although implementation of business rules within the domain layer is straightforward since it implements the processes, conventional approaches suffer from issues with scattering, tangling, and repetition of the rules across a whole layer as is stated in Chapter 1. It arises from relations between business contexts, which is discussed in Chapter 5.

The aspect-driven development focuses on the separation of the concerns and efficiently deals with repetition, scattering, and tangling of concerns as well as their transformation and automated reuse. Its implementation into design of the domain layer is intuitive and straightforward. Since violation of data consistency or violation of business policy may have significant impact on business, execution of business logic is very sensitive. In consequence, the ADD utilizes business rules to verify assumptions (preconditions) of business operations to protect them. The aspect weaver utilized by the ADD accepts a business operation and its context, and places a wrapper around the operation to automatically verify the assumptions on each call.

Although the benefits of utilization of the ADD in the domain layer are not as strong as in other layers, they are still significant. First, source code of business operations is less complex as an operation can assume both data and a context are valid. Second, the ADD decouples business rules and a place of their use to avoid manual repetition of business rules across multiple contexts, which is discussed in Chapter 5. This enables reuse of the rules across multiple contexts and in many places throughout a system. Third, utilization of the ADD in this layer enables full potential of the approach. Automated reuse of concerns throughout a whole system has major benefits, which are elaborated in Chapter 4. And finally, there are obvious benefits such as lower error-proneness, maintenance efforts, and involvement of domain experts in development. On the other hand, implementation of the ADD brings significant initial efforts and barrier. The need to implement DSLs and aspect weavers outweigh benefits in the domain layer as there exists alternative approaches with similar objectives. Comparing them to the ADD, they have much lower initial overhead and barrier. However, they are often limited to the domain layer, are clumsy ,and have insufficient expressiveness.

6.4 Data Layer: Querying of a Persistent Storage

An EIS is a system designed to maintain large volumes of data via business processes and expose it to users. Therefore, each EIS accesses a persistent storage, e.g., a database or a remote service, to be able to fulfill its purpose. In case of the layered architecture, which is used as an example in this thesis, the data layer, which is illustrated in Figure 1.1 and discussed in [42], facilitates access to data via read and write operations. The write operations directly manipulate data, and existing approaches such as ORM [6, 42] and the Active record design pattern [42], simplify their implementation. However, the read operations are the views meaning that they return data matching a filter, where the filter consists of business rules. Considering read operations within EISs, the filters or their parts significantly repeat across the operations, which leads to high error-proneness and difficult maintenance. Therefore, this section proposes utilization of the ADD to separate these repeated rules and have them automatically distributed at runtime. This utilization of the ADD addresses the problems stated in Chapter 1 and besides the common benefits discussed in Chapter 4, it reduces the repetitions in the data layer.

There exist various persistent storages. For example, there are remote services, files in a file system, NoSQL databases, and relational databases, which are the most common. Therefore, this section works with an EIS with the layered architecture and a relational database, but it seems that this utilization of the ADD is not specific to relational databases and could also be migrated to other storages.

Consider the request flow in Figure 6.1. Usually, there are two communication patterns between the domain and the data layers. First, the data layer implements a view on data as an operation and exposes its to upper layers. Second, the data layer exposes it internal API, usually encapsulated behind a facade [45], and lets the domain layer construct a view itself. In the latter case, the domain layer constructs a filter objects, which encapsulates the constraints, and passes it into the data layer. It transforms the constraints into an executable form, e.g., an SQL, and queries the storage, while in the former case, the filter object is hidden inside the data layer. Either way, a filter object or its alternative constraints the view[8].

Usually, the data layer facilitates various views on the same dataset and these views differ only in insignificant nuances, so filter objects has a subset of rules in common. Consider a `Product` entity from the running example defined in Section 1.1. There exist following three use cases:

- UC 2: List unavailable products
- UC 4: List free products
- UC 5: List products with free shipping

First, the UC 2 operates with all valid products, which are not in the stock. This operation is invoked by staff in order to restock unavailable products. Second, the UC 4 considers all free products, which are usually not buyable but are services free of charge automatically added into a buyer's cart, when he meets some requirements. Third, the UC 5 considers all products eligible to free shipping, i.e., products, which are expensive enough.

Although it seems that the simple examples above have nothing in common, consider the `validUntil` attribute indicating if a product is available or was already deleted[9]. Implementation of queries executed within these use cases is illustrated in Listing 6.1 using JPQL. Note the repeated constraint `validUntil is NULL`. Although it seems like minor issue in this simple example, scale it to production-size EISs with hundreds of queries and hundreds of entities. When the amount of repetitions becomes significant, it is also common to forget to put some constraints into a query, and the query tends to get obsolete due to errors in maintenance. In consequence, also this repetition of business rules in the data layer may cause significant issues discussed in Chapter 1. The rules are repeated, scattered, and must be synchronized with expectations in the domain layer, which put additional mental demands on developers.

[8] For example, there exists Hibernate Criteria API, which enables type-safe construction of such a filter object. Alternatively, when the data layer hides a filter, then a query is often manually constructed in a query language such as the Java Persistence Query Language (JPQL) [6].

[9] Production-ready EISs do not delete any data as it is the most valuable thing they have and its removal might result in missing references or information loss. In consequence, there exists a flag indicating whether an instance is deleted or not. This thesis uses the attribute `validUntil`.

Listing 6.1: JPQL queries into a relational database in the data layer

```sql
-- UC 2: List unavailable products
SELECT p FROM Product p WHERE p.inStock = 0 and p.validUntil is NULL;

-- UC 4: List free products
SELECT p FROM Product p WHERE p.price = 0 and p.validUntil is NULL;

-- UC 5: List products with free shipping
SELECT p FROM Product p WHERE p.price >= :limit and p.validUntil is NULL;
```

6.4.1 Utilization of the ADD

Since the data layer suffers from repetition and scattering of business rules, this section implements the aspect-driven development approach into design of this layer to deal with the issues raised from repeated concerns. First, there is established understanding of the rules, and then defined mapping into the aspect-oriented programming, which is an essential component of the approach.

Basically, a data retrieval is a business operation as it is possible to think about it in terms of use cases. For example, consider the *UC 2: List unavailable products*. The business domain defines preconditions, which must be satisfied to access this data. For example, a user must have a staff or an administrator role. The business domain also defines postconditions, which are valid for its output and, as defined in Chapter 5, both preconditions and postconditions create a business context, which is a description of an operation from a business perspective. Therefore, the constraints for data retrieval are postconditions in terms of the ADD.

Having a business context for each data retrieval operation in the data layer, the ADD enables its extraction into the repository and description in a DSL, which is elaborated in Chapter 5. Then, an aspect weaver intercepts a call of a data retrieval operation in the data layer, extracts its business context, and constructs a query. Next, the weaver passes the query to a provided executor to get data matching the filter. Finally, it returns the data to the caller. Note, that the data retrieval operation was not actually executed. The aspect weaver supplied the behavior instead [A.7].

However, the ADD is implemented into design of a component via formalization of the component in terms of the AOP, which is the underlying mechanism in the approach. Furthermore, the formalization provides guidance to actual implementation in source code as all important components are discussed.

An aspect in the data layer is a set of business rules, which compose a significant concern in an EIS, since it deals with multiple similar views over the same dataset. The views are defined via a set of constraints, i.e., business rules. As the views are often similar and share a significant part of the rules, they tend to be manually repeated among data retrieval operations.

Advice in the data layer is a business context and the data layer applies both preconditions and postconditions. While preconditions are optional as it might be safe to assume implementation of security in the domain layer, postconditions constrain a dataset and define a view. Therefore, there are two uses of business rules:

(a) When a business context referenced by a data retrieval operation defines its preconditions, then it is *input and context validation.* It is identical to validation in the domain layer, which is already defined in Section 6.3.

(b) Postconditions constrain a dataset and must be considered within a query into a persistent storage, which is a relational database in this example. Therefore, an advice in this concern is *a constructed query into a storage.* In case of a database, it is a query with unbound parameters and binding is done later. In consequence, assuming business contexts are immutable at runtime[10], the queries are safe to be prepared in advance to increase performance when an operation is called. So, for each context, postconditions are extracted and vertically transformed into a query language, and later bound with an execution context.

A joinpoint in the data layer is a single injection point in the request flow:

① The joinpoint *around a data retrieval operation* wraps a body of each data retrieval operation in the data layer, which links a business context in order to replace it by an aspect weaver.

In case of a data retrieval operation, which only queries a persistent storage for data matching a given filter, then this operation can be substituted by automatically generated implementation. Therefore, a body of such an operation is nested in a block, which is the joinpoint and can be replaced by aspect weaving. Considering common terminology, e.g., used by AspectJ [66], it expects the *around* advice, which controls invocation of the wrapped body.

Pointcuts in the data layer apply a business context to each data retrieval operation referencing it, in order to instrument the aspect weaver.

Aspect weaving in the data layer substitutes implementation of data retrieval operations, which gather data from a persistent storage. When the operation links a business contexts, which defines both preconditions and postconditions, then there already is a fully defined behavior and its further manual implementation is not needed. Therefore, the aspect weaver extracts the business context of the operation, and constructs a query considering postconditions [A.7]. Then, when the operation is invoked, it binds the query with parameters within an execution context and calls an available executor, which is responsible for actual execution of the query. Finally, it returns acquired data. Similarly, the aspect weaver intercepts a call to the operation and validates its input, as already discussed in Section 6.3.

[10] It is not necessary as discussed in Section 6.5. However, it is immutable most of time.

In conclusion, implementation of the ADD into design of the data layer saves significant amount of code and efforts, since the aspect weaver replaces data retrieval operations, and produces executable code by itself. Besides the benefits of the ADD already discussed in Chapter 4, its utilization in the data layer is complementary to other components of a system. Without additional efforts, another reuse of already maintained business rules greatly increases overall efficiency.

6.4.2 Summary

The data layer of enterprise information systems silently suffers from scattering and repetition of business rules, which results from manual construction of queries into a persistent storage. Although it might not seem significant, scaling it into a large system with hundreds of entities, views, and queries, the repetitions significantly pollute source code. Eventually, their maintenance becomes very error-prone and tedious due to high amount of scattered repetitions. The ADD recognizes business rules in the data layer among cross-cutting concern, and utilization of this approach focused on the separation of concerns significantly improves maintenance efforts of this layer.

Although conventional EISs often operate with complex queries aggregating data over multiple resources, it is a best practice to store these complex queries in a database or other storage as an optimized view. Then, a system produces only simple queries, which can be automatically constructed by a framework, while complex or performance-sensitive queries can be left for manual optimization. Therefore, considering a simple operation for data retrieval, it is possible to define a business context of the operation and put it down into the repository in a convenient DSL tailored for business rules. Then, the source code of the operation is replaced by its signature and a reference to that context to instrument an aspect weaver to generate the body of the operation. The weaver extracts postconditions from the context and composes them into a query, which is on demand bound with parameters from an execution context and executed.

Although utilization of the ADD increases the initial barrier and requires significant amount of efforts to implement the weaver, its complementary benefits outweigh these disadvantages when the ADD is also used in other components of a system. With minimum additional efforts, the ADD reuses already maintained business rules in another component, i.e., the data layer, and facilitates data retrieval. As a result, there is no need to manually maintain repeated and scattered conditions in queries as well as there is no need to manually compose them, if they are not complex or performance-sensitive. In addition, it is possible to reuse the validation mechanism from the domain layer to validate input of operations. In conclusion, although the ADD requires major initial efforts, the tools such as the weaver and the DSL for business rules are not project-specific, thus are reusable across multiple projects, which significantly utilizes the initial investment.

6.5 Service-oriented Arch.: Reuse of Concerns

The previous paragraphs elaborate cross-cutting concerns in an EIS, and in order to eliminate their significant scattering, tangling, and repetitions, and to reduce negative impact of the concerns on development and maintenance of a system, several implementations of the ADD into design of existing components in the layered architecture are proposed. The ADD is a high level approach tailored to deal with cross-cutting concerns in EISs via the separation of the concerns. However, the number and complexity of requirements of EISs grow beyond limits of a single system. Therefore, contemporary EISs often utilize service-oriented architectures (SOAs) or its derivations such as Microservices[11] to decompose a system into smaller maintainable components (services). In consequence, an EIS is split up into multiple smaller but tangled systems. This significantly limits reuse of cross-cutting concerns proposed by the ADD. In conclusion, despite utilization of the approach in a single system, there exists significant repetition of concerns across multiple services. This section discusses implementation of the ADD into design of services in the SOA in order to facilitate cross-service reuse of cross-cutting concerns, especially but not limited to business rules.

Consider the example in Figure 1.3 in Section 1.1, which demonstrates a small e-commerce system, scaled into a large production-size system maintaining large volume of data and having hundreds of other services. Then, size of the system, and coupling among services cause significant issues and impair further evolution of the system. Therefore, such a system is often migrated into the SOA, where each component is implemented as a standalone unit exposing its API and depending on other existing services. Then, the Figure 6.4 illustrates the running example migrated into the SOA. As the original components in the domain layer follow branches of business, then they become services. However, besides this large-scale architecture, each service has its own internal, possibly the layered architecture, which is used as the example throughout this thesis. Moreover, while the services implement business logic, there exists a service implementing a facade to internal API and exposing the UI to end users.

The example above deals with two types of services. First, a *standalone service*, which does not have any dependencies, and maintains its own data and exposes its API. Second, a *composite service*, which implements business logic over one or multiple services it depends on. For example, `User Service` and `Product Service` are standalone services because they maintain primary data and do not have any depen-

[11] While the Microservices and other derivations also deal with scaling of distributed systems, they have different assumptions. Considering the scope of this work, the most significant difference is usually in orchestration versus choreography. This work is agnostic to both approaches, but delivers better benefits with choreography, which is implemented by Microservices [30, 44]. Although the proposed implementation of the aspect-driven development should also apply to other architectures, since there exist other differences, this thesis leaves it for future work and focuses on the SOA instead.

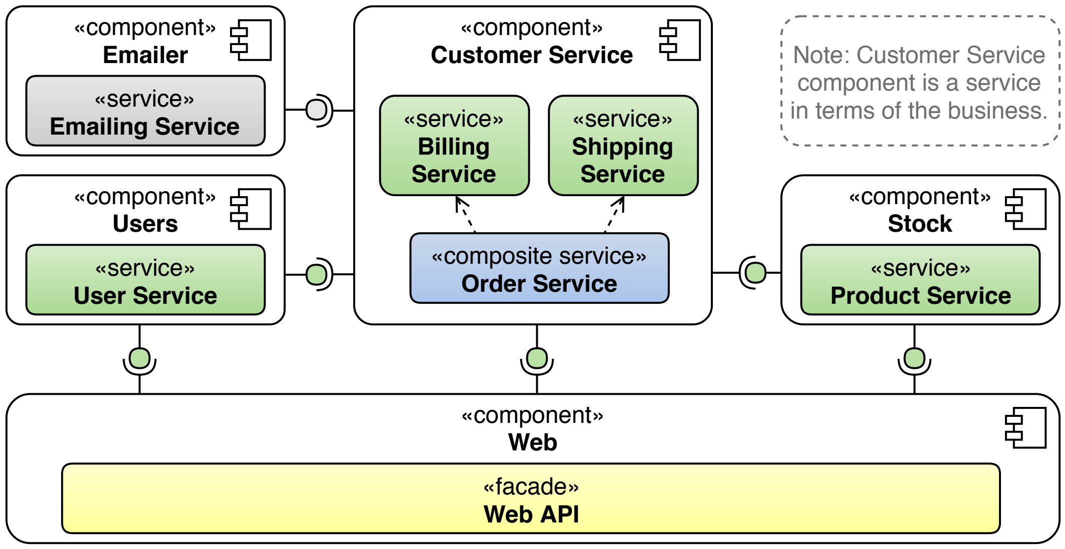


Figure 6.4: Example e-shop system in the service-oriented architecture

dencies. Contrary, `Order Service` depends on both `User Service` and `Product Service` because each order tracks a customer and a list of ordered products.

Considering business rules as a significant concern, implementation of a standalone service is straightforward and may fully benefit from the ADD implementation. Unfortunately, a composite service faces the challenge of limited inspection of transitive business contexts, i.e., limited reuse of business rules declared by services it depends on. This limitation results from difficult concerns extraction because, in general, each service can be implemented on a different platform using different technologies. Since extraction of concerns from source code is challenging and limited as discussed in Chapter 3, the ADD takes place to ease this extraction in addition to other benefits of the separation of concerns. For example, consider a business operation for registration of a new user implemented by `User Service`. It defines and maintains its own business rules including validation rules itself. However, when a user is registered during creation of an order, which is a common use case, `Order Service` has to accept only a valid input to be able to further process the registration request and delegate it to `User Service`. Unfortunately, since `Order Service` is unable to reuse business rules used by `User Service` using conventional development, it usually results in manual repetition of business rules in order to facilitate this behavior.

In consequence, while the SOA efficiently deals with some issues of large-scale systems, there exist some significant challenges. Considering composite services, cross-service reuse of concerns is one of them. Fortunately, while the ADD efficiently deals with the separation and automated reuse of concerns in a single EIS, this section proposes its implementation into design of the SOA. Besides the common benefits of the ADD, the proposed implementation enables automated cross-service reuse of concerns, which positively impacts both communication in a development team and development efforts. Furthermore, it opens new possibilities such as centralized management of configuration and business rules without any centralization in the source code.

6.5.1 Utilization of the ADD

While the ADD is designed for a single EIS to separate the concerns and automatically weave them back together at runtime, this section discusses implementation of the ADD into design of services in the SOA. This section formalizes the domain of the service-oriented architecture in terms of the AOP, which is significantly utilized by the ADD, and concludes with new possibilities available with the ADD for the SOA.

The repository is the essential concept of the ADD, since is introduces the single point of the truth. However, the SOA, Microservices, and other derivations focus on decomposition of a business domain and an entire EIS into many smaller standalone units, which prevents existence of such a place. Therefore, to overcome this limitation and to benefit from the ADD, it is necessary to construct a virtual runtime repository, which aggregates the concerns also from other services. In consequence, to fully benefit from concerns reuse, all services in the SOA must implement the ADD to be able to separate and expose their concerns.

Consider `Order Service` and the registration of a new user during order processing, as discussed above. While each service implements the ADD and therefore each business operation defines its business context, it is safe to assume there exists an operation creating a new user within `User Service`, and subsequently a business context of that operation. Therefore, `Order Service` assumes existence of this context although it does not know its implementation at compile time. However, it may consider it external and include this context via context composition at runtime.

> **Definition 6.3** *An external business context is a business context defined and maintained by another service.*

Fortunately, although the context is not known at compile time and a static single place of the truth, i.e., a place in source code, cannot exist, it is possible to define the repository at runtime because all concerns are immutable. Furthermore, it is not necessary to create a single know-it-all repository encapsulating all concerns in the entire SOA because the repository would be too complex and there is no need for it. Instead, each service creates its own repository at runtime and fills in all external concerns from other services, which replaces the static repository defined in Chapter 4.

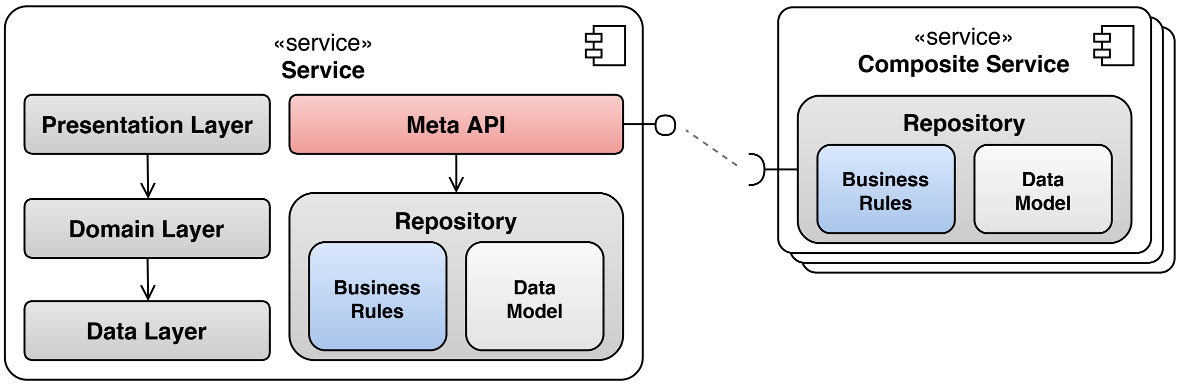


Figure 6.5: The layered architecture of a service with the ADD for the SOA

> **Definition 6.4** *The runtime repository is a single point of the truth within a service in the SOA and extends the static repository with implementation of external concerns fetched from other services.*

Finally, when a concern in a service includes another external concern, i.e., a concern of another service, it is essential to expose the external concerns maintained in the remote repository via API. However, a security aspect and encapsulation step into the process and thus not all concerns are supposed to be exposed. Therefore, only parts of concerns, which are supposed to be reused by other services, are published via API of the repository. For example, only some business contexts are supposed to be exposed to other services, while other are intended only for internal use. Consider a registration of a user. While a business context of an operation in the domain layer is used only within the service, a business context of an action handler in the presentation layer is intended for communication with other services and thus is exposed.

> **Definition 6.5** *A public concern in a part of a concern, which is exposed via API of the repository for reuse by other services.*

Although the runtime repository fully replaces the static repository from the ADD definition in Chapter 4, the implementation of the ADD into design of a service is similar to implementation of the ADD into design of a single standalone EIS. The most significant change lies in the introduction of external concerns and the runtime repository, which slightly modifies the architecture of given service, but it bridges the gap between services and enables cross-service reuse of concerns. For example, consider the conventional layered architecture [109] accommodated to needs of the ADD for the SOA, which is illustrated in Figure 6.5. Just like with the conventional ADD, a service separates its concerns and stores them in the repository, which helps to decompose the system into smaller units with single responsibility. However, each service introduces the meta API exposing public concerns or their parts for other services.

While it is not mandatory, it is beneficial when also other services implement the ADD. For example, consider that although the implementation of an external context is not known during implementation of a service, it is safe to assume that it exists. But when the other service does not implement the ADD, it is not possible to reuse the context or it might not even be defined. In conclusion, when some services do not implement the ADD, the overall efficiency of the ADD in the SOA is reduced.

Although the previous text discusses limitations of the conventional ADD and its migration into the SOA, it is important to formally define mapping of SOA components into terms of the AOP. The mapping suggests implementation of the ADD, which significantly utilizes the AOP, therefore, besides the formal definition of a viewpoint, it also provides a guide for implementation.

An aspect in the SOA is a concern, which cross-cuts across multiple services and tends to scattering, tangling, or repetition. Therefore, considering mutual independence of services, varying platforms and technologies, and possibly even different development teams, manual maintenance of such a concern is significantly error-prone, tedious, and increases development and maintenance efforts and possibility of inconsistencies. In the SOA, this thesis recognizes two cross-cutting concerns:

(i) *Business rules*, more specifically a business context of a business operation, define *preconditions*, *postconditions*, and *business domain configuration*. Operations of composite services often reference business contexts, or their subsets, of services they depend on. Consider `Order Service`. Besides reuse of validation rules, which is illustrated above, the service references configuration of `Billing service` to reuse a constant defining VAT rate to properly compute the price.

(ii) *Data model*, which is a structure of the model in the protocol, is always considered on both sides of the communication to serialize and deserialize the data. This aspect can be used for verification that both communicating services expect the same protocol structure and there are no inconsistencies.

Advice in the SOA is functionality to be weaved in. Although there are two aspects, the ADD organizes business rules into business contexts, which have multiple parts and thus multiple uses. Therefore, there are following advices:

(a) *Business context preconditions* advice is a set of assumptions to meet before a business operation is executed, e.g., the user is logged in and has the required privileges. The rules are transformed into a platform-specific executable form as they are evaluated in the beginning of an operation.

(b) *Business context postconditions* are rules applied after execution of a business operation, e.g., data filtering based on the user's privileges or expected results. The rules are transformed into a platform-specific executable form as they are applied at the end of an operation.

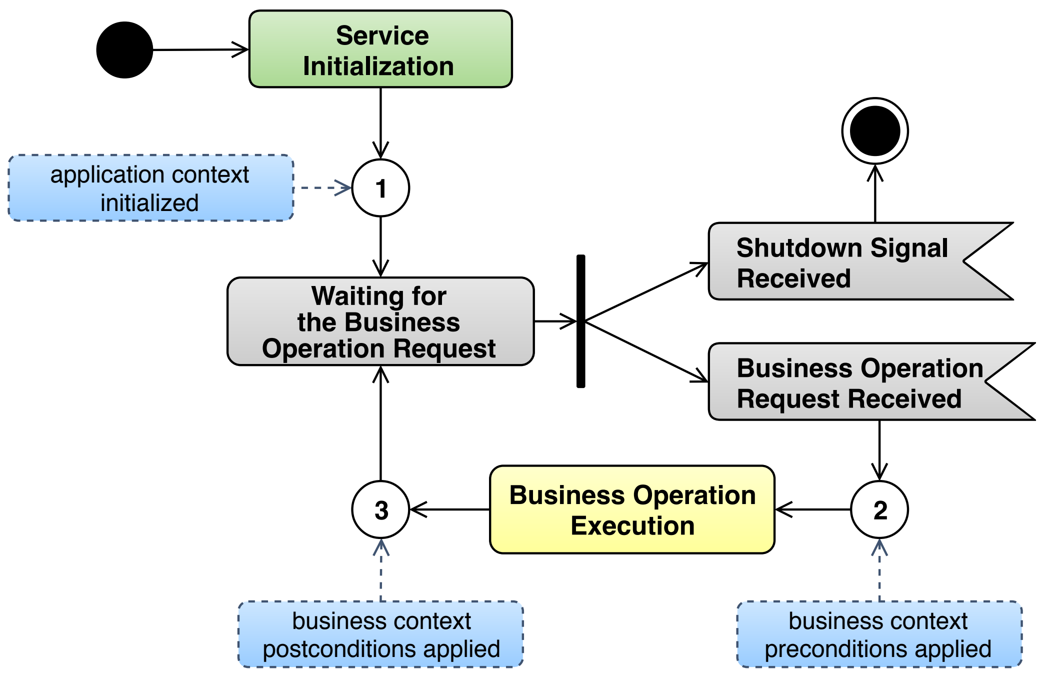

Service Initialization
application context initialized
1
Waiting for the Business Operation Request
Shutdown Signal Received
Business Operation Request Received
Business Operation Execution
3
2
business context postconditions applied
business context preconditions applied

Figure 6.6: Life-cycle of a service and application of advices

(c) *Business domain configuration* is a part of the application context represented by a key-value map of business domain-related constants used within a service, i.e., during rules evaluation or business logic execution, e.g., the VAT rate.

(d) *Data model* advice contains information about the structure of public business objects defined within each service, i.e., a name of objects and list of public fields and their types. As this information is used for validation of a protocol, the representation of a concern is not important.

Joinpoints in the SOA are places in the lifecycle of a service (dynamic joinpoints) available for a concern to be weaved in, which is denoted in Figure 6.6. There exist three following dynamic joinpoints:

① First joinpoint triggers during *initialization of a service*, when the service establishes its application context. Since some business contexts may reference external contexts, which are unknown at compile time, their implementation is fetched at runtime before the service is up and running.

② *Before the execution* of a business operation, it validates preconditions of the addressed business context, which is already defined in Section 6.3.

③ *After the execution* of a business operation, it applies postconditions of the business context, which is already defined in Section 6.4.

Pointcuts in the SOA select a subset of joinpoints for each aspect applying a particular advice. As Figure 6.6 suggests, the initialization of a service considers a data model to validate protocols of remote services, and external business contexts to fetch configuration of a business domain, which must be known before the service is running. Then, as defined in Sections 6.3 and 6.4, preconditions of a business context are considered at the beginning of an operation, while postconditions are applied when execution of the operation has finished.

Aspect weaving in the SOA combines all the advices into proper joinpoints at runtime with the respect to a current execution context, which is conducted by platform-specific aspect weavers. First, when a service starts, it initializes its application context including environmental variables and all business contexts. In an environment with shared business contexts such as the SOA, the service fetches external contexts from services it depends on. For example, *`Order Service`* requires business contexts and domain configuration defined by the *`Billing`*, *`Shipping`*, *`Product`*, and *`User`* services. Therefore, the service discovers the other services[12], downloads the external contexts, merges them into its application context, and exposes them in the repository. Finally, it extracts metadata of its public model and stores them within the repository as a public concern. Then, it verifies the structure of the communication protocol comparing its own metadata to the metadata of its dependencies.

Second, once the service is initialized and running, it expects requests to execute business operations. For each business operation, there is a business context defining the preconditions and postconditions, and when execution of a business operation is requested, the aspect weaver intercepts the call, checks the business context, and validates the applicable preconditions with the execution context. For example, it verifies the user is logged in. If the validation fails, the execution ends and the service returns an error. Similarly, when the execution of the operation finishes, the aspect weaver intercepts the response and applies the postconditions to restrict the returned data.

6.5.2 Summary

There exist many opened challenges in the SOA. For example domain decomposition, service discovery, composition, deployment, and evolution, inter-team communication, and the separation and reuse of the concerns across services. Unfortunately, the cross-cutting concerns impact both standalone EISs as well as distributed systems built on the SOA. While the separation and reuse of concerns within a standalone EIS is the same as within a single service, cross-service reuse is more challenging as each service may use different platform, technologies, and be built by a different development team.

The ADD is an abstract approach designed for the separation and reuse of the concerns in standalone EISs and is agnostic to implementation details. While it efficiently deals with the cross-cutting concerns in an EIS, it reaches its limits when it is deployed into the SOA. Therefore, the ADD for the SOA extends the original approach

[12] There exist various approaches to link services, e.g., a Service catalogue or an Enterprise Service Bus, but the ADD is agnostic to implementation details.

and introduces new elements such as public concerns, external contexts, and meta API, in order to enable cross-service reuse of concerns. Then, a composite service is able to automatically fetch and reuse external concerns from its dependencies at runtime, which significantly eases composition of services. Besides the common benefits of the approach, the ADD for the SOA also enables validation of a communication protocol, and reuse of business domain configuration, which a part of a business context.

In conclusion, similarly to the conventional ADD, the ADD for the SOA delivers significant maintenance improvement, codebase reduction, and context-awareness to services. It isolates cross-cutting concerns into the single point of the truth, and weaves them back together at runtime with respect to a current execution context. Use of DSLs enables involvement of domain experts in development. Automated distribution and reuse of the concerns at runtime remove the need for manual synchronization of all places, which mitigates maintenance efforts and lowers the risk of a human error. On the other hand, the ADD itself introduces significant overhead, as it requires design and implementation of DSLs, and platform-specific project-independent aspect weavers. Since the SOA usually utilizes multiple programming languages and platforms, it increases the number of required weavers. Therefore, development of the technological stack requires major efforts. Furthermore, all services in the SOA have to follow the ADD for the SOA, otherwise the reuse of concerns is limited.

This section introduces the ADD for the SOA. While its use opens new possibilities, this thesis leaves them for future work. For example, having all metadata exposed via API, it seems possible to maintain business rules for all services from a single place, e.g., a maintenance application, while their source code remains distributed throughout the services. Such an application can be managed by domain experts and furthermore, the changes to business rules and subsequently business contexts can be streamed throughout the SOA at runtime without any need of restart or even recompilation of any service. Further, it is possible to visualize the relations between services and perform some data mining and reverse engineering over available metadata. And finally, while the SOA is a generic architecture of distributed information systems, more evolved derivations exist such as Microservices and other architectures. While it seems that they would also benefit from the separation and reuse of concerns and the ADD for the SOA, the proof of this hypothesis is left for future work.

6.6 Summary

Contemporary EISs suffer from concerns tangling and scattering due to inability to separate the concerns. High information repetition, which leads to inconsistencies in source code, is among the most significant reasons behind highly error-prone and difficult maintenance of a system. The ADD, which is a high level approach proposed in this thesis, emphasizes the separation of the concerns and utilizes various existing approaches to introduce a single point of the truth and facilitate automated transformation and distribution of the concerns throughout a whole system (Objective 1). However, the ADD as an abstract approach does not enforce nor suggest any implementation details. It is generic and applicable into various environments. Therefore, this chapter elaborated multiple implementations of the ADD into design of common components, in order to demonstrate benefits of the approach comparing to conventional development (Objective 3). Although to properly implement the ADD, each implementation is formalized in terms of the AOP, since it is significantly utilized.

The aspect-driven development approach considers a domain model and business rules as a significant concern, which arises from the purpose of a system (Objective 2). And, as the ADD modifies an overall architecture of a system, there are various components, which may benefit from it. Considering a layered architecture, which is an example considered in this thesis, there are three basic layers: the data layer, the domain layer, and the presentation layer. This chapter demonstrated reuse of business rules and the model in the data layer, where it facilitates automated input validation and construction of queries into a persistent storage to provide filtered data matching postconditions in a given business context. Next, the domain layer also benefits from input validation in order to protect data and application logic. Contrary, the UI as possible implementation of the presentation layer benefits from the ADD much more. Since in the UI many concerns tangle together including UI layouts, UI widgets, and localization, implementation of the ADD into design of the UI is more complex but brings greater benefits. This chapter showed composition of UI forms as well as conditional rendering of a component based on a business context of another business operation. Furthermore, this chapter discussed modification of the ADD to support distributed multi-platform UIs with native applications, which use both shared and platform-specific concerns. Since complexity of contemporary EISs grows behind limitations of a single system, they are often decomposed into multiple services and organized into the service-oriented architecture or its evolved derivations. This chapter presented implementation of the ADD into the SOA providing both intra-service and inter-service reuse of the concerns, which enables central management of the concerns.

In conclusion, while the ADD is not specific to any architecture or environment, this chapter presented various benefits raised from the separation of concerns (Objective 3). The examples illustrate implementation of the ADD into design of different components of an EIS as well as different environments. However, the list of examples is not exhaustive. Considering other components, architectures, and environments suffering from concerns tangling and scattering, each may benefit from implementation of the ADD. Therefore, this chapter demonstrates utilization of the ADD, and opens a discussion over its additional use, in order to reduce tangling and scattering of the concerns (Problem 1) over multiple technologies (Problem 2), remove inconsistencies and repetitions (Problem 3), and ease maintenance of source code (Problem 5). Moreover, since the approach supports context-aware decisions (Problem 4), it eases implementation of modern context-aware applications supporting various mobile devices and runtime contexts. Unfortunately, although utilization of the approach introduces major benefits, its deployment into an EIS carries significant initial overhead. While use of multiple DSLs increases the mental barrier, the need for implementation of various aspect weavers stands for a technological barrier. Fortunately, both domain-specific languages and aspect weavers are not project-specific, therefore can be reused across multiple projects, which significantly utilizes the efforts.

Part III

Results

Chapter 7

Case Studies and Evaluation

The aspect-driven development is an approach to design and development of EISs introduced in previous chapters. While contemporary systems usually suffer from significant concerns tangling, scattering, and information repetition, the approach addresses these issues via utilization of multiple existing approaches and the separation of concerns (Objective 1). Complexity, abstractness, and claimed benefits of the ADD require further work in order to demonstrate and evaluate the approach (Objective 3). The evaluation can be divided into 3 steps:

1. Providing a proof of the concept to demonstrate the novel architecture and showing that it works and delivers claimed benefits.
2. Evaluating the approach in contrast to alternative existing approaches and highlighting differences including benefits and limitations.
3. Conducting a large case study evaluating the approach in real production-size system, measuring its efficiency, claimed benefits, and evaluating its limitations.

Therefore, complete evaluation of the approach is a challenge itself, and, unfortunately, goes beyond the scope of this thesis. However, the first two steps are well covered in published researched papers written together with this dissertation thesis. The papers elaborate topics from the previous chapters in depth and always include either a case study comparing a conventional approach to the implementation with the ADD, or a proof of the concept, which illustrates use and benefits of this approach. Unfortunately, conducting a large case study requires significant amount of work, production-ready tools[1], and cooperation with a business company that would provide access to a real system, therefore it is left for future work. Although, for the sake of completeness of this thesis, this chapter summarizes the most significant case studies conducted this thesis, compares the approach to other existing alternatives, and illustrates implementation of the ADD in a proof of the concept implementation of the UI.

[1]It includes design of all DSLs and implementation of all supporting tools such as parsers, editors, and development tools. It also requires implementation of aspect weavers for all components, environments, and technologies, which might be a challenge itself.

While the ADD brings significant benefits, there are various limitations reducing its efficiency. As it uses multiple DSLs to describe concerns and aspect weavers to weave them together at runtime, there exist major initial overhead and both technological and mental barrier. However, to demonstrate and evaluate the approach, its benefits, and limitations, it is not necessary to have production-ready tools. Even stubs and partial implementations show intended behavior and qualities. Therefore, each research paper published together with this thesis includes its own case study. Unfortunately, the papers were published before this thesis was written, therefore they do *not use the running example* introduced in the first chapter of this work. Each paper describes its case study to evaluate, instead. Even though, the ADD implementation discussed in Chapter 6 is well covered by case studies:

(i) Concerns tangling in the UI, their decomposition, and automated runtime weaving utilizing the AOP is discussed in [17, A.3]. Therefore, it is sufficient to extend that research by business rules, which are a significant concern in an EIS and not addressed in that papers. This extension and case study are in [A.13].

(ii) Implementation gets more complex considering the distributed multi-platform UI, where multiple native applications participate backed by RESTful API. As is stated in the previous chapter, the implementation of the UI is similar to the previous case, although the concerns are exposed via the API. This research introduced in [A.5] and extended in [A.1] also brings a case study comparing conventional development to utilization of the ADD. Furthermore, server load and benchmarks are evaluated in [A.12].

(iii) Reuse of concerns for input validation in the domain layer is presented in [A.15] and formalized into the preliminary version of the ADD in [A.8], which shows a comparison of the ADD in the domain layer and a conventional development.

(iv) Reuse of concerns in the data layer is elaborated in [A.7], which transforms business rules into SQL queries, and evaluates a proof of the concept implementation.

(v) The SOA deals with more challenging concerns tangling, scattering, and information repetition. The research in [A.4] presents the adjusted ADD enabling inter-service concerns reuse, as is elaborated in the previous chapter. The paper demonstrates a conducted case study, which compares a conventional design of services to services implementing the ADD.

(vi) Finally, transformation and reuse of the concerns into business documentation is published in [A.6]. The paper also discusses parameters of a case study and presents preliminary results illustrating intended use and output [A.2].

In conclusion, the first step of the evaluation is well covered in the published papers. Furthermore, besides these focused case studies evaluating a specific component of the architecture, along with this research, additional case studies there were conducted comparing the ADD to alternative existing approaches [A.14, A.9, A.11]. These case studies implement an identical system using various approaches, measure their efficiency, and discuss maintenance efforts. For the sake of completeness, the following sections discuss the most significant results of these papers to summarize the evaluation of the ADD. Although, for additional details proceed to reading the original papers.

7.1 User Interface: Proof of the Concept

A proof of the concept illustrates intended behavior, claimed benefits, and suggests possible limitations. Its main purpose is to demonstrate that the idea is achievable. Therefore, there are no requirements on production-level tools or implementation. Partial implementations and stubs work fine, if they serve their purpose. Considering the ADD and this thesis, the UI is the most challenging component. Most concerns tangle together in the UI, scatter throughout a whole component and even through lower layers of a system. It results from linearization of orthogonal concerns into source code, as is discussed earlier in this work. The separation of concerns in the UI is the easiest to observe and delivers maximal benefits. In consequence, this section demonstrates possible implementation of the ADD in the UI, to suggest intended use and benefits.

Development of the UI suffers from various challenges, concerns tangling is one of them. There exist various attempts to decompose the concerns to overcome this challenge, which is elaborated in Section 3.4.3. The approach presented in [17] already followed a similar idea, although it is not that general. However, it utilizes the AOP to decompose most of the concerns and weave them back together at runtime, which was extended and formalized in [A.3]. Unfortunately, it does not consider business rules. In conclusion, it is sufficient to demonstrate reuse of business rules from the repository together with the UI constructed using a technique presented in [A.3].

Consider an issue tracking system as an example[2]. The repository maintains these concerns: 1) UI layouts, 2) UI widgets, 3) business rules, 4) localization, and 5) a data model, which is not located in the repository but is extracted from source code instead. Utilizing the research[3] from [A.3], representation of the concerns is straightforward.

(i) Listing 7.1 demonstrates a template for a two column form layout, i.e., form fields are organized into two columns. The template is written in HTML extended with a custom tag and expressions in the `af` namespace. The tag indicates repeatable block, which is repeated until all form fields are rendered. The expressions are placeholders to be replaced by form fields [A.3].

(ii) Listing 7.2 illustrates a simple template for presentation of a textual field, i.e., a `String` field. The template is written in HTML for JavaServer Faces 2.0 [28] and

[2] The case study was already published in [A.13] and uses a different example than this thesis.

[3] The research in [A.3] is implemented in an open source framework AspectFaces, which extends JavaServer Faces and facilitates runtime forms composition. It is available at aspectfaces.com.

Listing 7.1: Two column layout template

```
<table class="two-column-layout">
  <af:iteration-part>
    <tr>
      <td>$af:next$</td>
      <td>$af:next$</td>
    </tr>
  </af:iteration-part>
</table>
```

Listing 7.2: Example of a UI widget for AspectFaces

```
<jsf:inputText
  id="#{prefix}$field$"
  label="#{text['$entity$.$field$']}"
  validate="$businessRules.toJS()$"
  value="#{instance.$field$}" .../>
```

uses various meta-information provided within an execution context. While this example is not complex, implementation of the AspectFaces provides additional various configuration options [17].

(iii) Consider an operation to report a new issue. Then, Listing 7.3 shows a business context[4] of this operation. It considers length of a title and a description, checks priority range, and verifies that the issue defines its type. While the context is maintained in the repository, it is reused in 1) the domain layer to validate the input of the operation, 2) the server-side of the UI to validate the input of the action handler, and 3) the client-side of the UI, where it is used for input validation directly in a UI form.

(iv) The localization is maintained as localized key-value pairs, where a key is an internal codename and the value is its localized translation into a particular language mutation. For example *issue.title: Title*.

(v) Finally, a data model is a simple POJO Java bean, which backs the input form. Each field has its name and type, and unless it is marked as it is supposed to be ignored, it is extracted by the aspect weaver in order to compose the form.

Having the concerns represented in source code, then data visualization is trivial. Listing 7.4 illustrates use of the aspect weaver in order to compose a form to report a new issue. The *af:ui* component accepts a model of a form, a business context, and a layout to be used. While the model is a new issue and the context defines its validation rules, the layout is extracted from an execution context, which contains a suggested layout for user's device. However, to properly render a form, input fields must be mapped to UI widgets, i.e., mapping of joinpoints and advices, which are

[4] The case study in [A.13] uses JBoss Drools language. However, the example was migrated into the language proposed in Chapter 5 in order to make it consistent in the thesis.

Listing 7.3: Example of a business context: Report an issue

```
context: Report an issue
  inputs:
    issue: Issue
  preconditions:
    issue.title not empty
    length(issue.title) is greater than or equal to 15
    length(issue.title) is less than or equal to 300
    issue.title matches "^[a-zA-Z0-9 ]*$"
    length(issue.description) is less than 1000
    issue.priority is between 1 and 3
    issue.type is required
```

Listing 7.4: Use of AspectFaces to compose an input form at runtime

```
<af:ui instance="#{issue}" layout="#{device.layout}" context="
    Report an issue"/>

<!-- context-aware action -->
<jsf:button
  value="Delete the issue"
  rendered="#{context(bean,'deleteIssue',issue).isSatisfied}"
/>
```

pointcuts. Example of the mapping is in Listing 7.5. For each data type, the mapping considers a default UI widget and additional widgets, which are used if a given condition is satisfied. For example, there exists different widget for a password, even though it is also a *String*. The conditions expect certain implementation of a data model inspector, which must inspect and publish all additional meta-information. For example, a password is marked with *@UiPassword*, and the marker is exposed into the mapping as a *password* variable.

Similarly, Listing 7.4 also demonstrates conditional rendering of a button to delete an issue. It is rendered only if a business context of the operation *deleteIssue* is satisfied, which might and should include security check to prevent unauthorized users to invoke the operation. Reuse of a business context also within a button greatly contributes to consistency of business constraints within an EIS.

In conclusion, the example presented in this section covers all aspects of the automated composition of input forms in the UI. Having each concern described independently in the most convenient, domain-specific language results in the best possible performance and effortless maintenance [59]. Furthermore, automated reuse of busi-

Listing 7.5: Mapping of data types of input fields to UI widgets

```
<type>String</type>
<cond expr="${password==true}" tag="passwordTag.xhtml"/>
<cond expr="${link==true}" tag="linkTag.xhtml"/>
<cond expr="${maxLength>255}" tag="textAreaTg.xhtml"/>
<default tag="textTag.xhtml"/>
```

ness rules throughout a whole system removes the need for their manual synchronization, which avoids inconsistencies in source code and reduces maintenance efforts [A.3]. Considering the UI, there are no validation rules manually written in HTML or in Javascript. They are fully provided by the server, transformed from the platform-independent format into JavaScript, and bound to UI events to deliver good user experience. Comparing the source code to the conventional alternative, there are a significant code reduction and easier maintenance, which confirm what was expected [A.13]. Unfortunately, to have the ADD production-ready, it requires production-ready DLSs and aspect weavers, which significantly increases initial overhead [A.8].

7.2 Distributed User Interface: A Case Study

In general, the objective of a case study is to compare various approaches, measure their qualities, and formulate results. Considering the ADD, a case study is conducted in order to compare this novel approach to existing alternatives. The study can focus either on a specific part of a system or compare approaches in general. As general comparison requires significant amount of work and production-ready tools, it is left for future work. Case studies conducted within this theses focus on a part of an architecture and evaluate approaches dealing with a specific challenge. Considering the UI from the previous section, decomposition of the concerns possibly significantly reduces amount of source code, as it removes many manual repetitions. Furthermore, maintenance of such code should be much easier, as each information is managed only once and in a single place, therefore there is no need to search for all occurrences and deal with inconsistencies. This section describes a case study conducted in order to evaluate efficiency of the distributed UI, which is implemented for multiple platforms and backed by a RESTful API at a server.

Implementation of the UI of an EIS is a challenge regardless of the target platform as that is where many concerns tangle together. Although, considering the distributed multi-platform UI, the number of concerns grows up for platform-specific concerns. Therefore, there is more to consider as well as more to implement, as each view must be implemented for each platform, which obviously requires significant amount of manual code repetition. Fortunately, the ADD proposes a technique to reuse platform-independent concerns and automate composition of input forms. In consequence, this case study demonstrates reuse of both platform-independent and platform-specific concerns streamed from a server into a client-side application implemented for two platforms. The study compares the conventional approach to a system following the ADD.

Consider a small issue tracking from the previous section. There exist one server application with RESTful API implemented in Java and two clients: a web application as a JavaScript rich client, and a native mobile application in Swift for iOS platform.

Table 7.1: Representation of UI concerns in the multi-platform UI

Concern	Platform	DSL
Data model	-	Java
Bussiness rules	-	JBoss Drools DSL[6]
UI Layouts	JavaScript	HTML
	iOS	JSON
UI Widgets	JavaScript	HTML
	iOS	JSON
Localization	-	properties in plain text

Then, consider the following scenarios: 1) a user reports an issue, 2) a developer resolves the issue, and 3) an administrator creates a project. Each of these represents a form-oriented context-aware view in both clients. In these scenarios, the case study considers 15 business rules, e.g., description of the issue must be between 15 and 300 characters, 1 iOS and 3 JavaScript layouts, and 6 UI widgets for each platform.

The system is implemented twice, i.e., using both the conventional approach and the ADD with the separation of concerns. There are several DSLs describing the concerns, which is summarized in Table 7.1. Although involvement of multiple DSLs introduces initial overhead, their efficiency significantly increases their expressiveness and reduces SLOC[5]. Nevertheless, besides the business rules, common programming and markup languages are used to reduce the overhead.

Having two implementations of a system enables comparison of source code efficiency. Table 7.2 delivers SLOC comparison of the conventional approach to the ADD implementation. The ADD separates 5 UI concerns and the domain model in Java overlaps with the conventional approach in lower layers of the system. The system with the ADD describes the UI for a web application in 197 lines of HTML, 256 lines of localization statements, 94 lines of business rules in JBoss Drools, and 121 lines of JSON configuration defining the UI the iOS native client. Besides these, it requires 271 lines of code in total to configure both clients. In consequence, it makes 939 SLOC in total, which is 45 % SLOC reduction comparing to the conventional approach [A.1]. Next, consider the size of this case study and a low scenario overlap. Imagine, how the reduction would scale up in much larger systems with multiple similar, more complex, and context-aware views. The savings might be very significant.

Unfortunately, in addition to the application-specific SLOC, the ADD requires implementation of reusable, project-independent, but platform-specific aspect weavers. For this study, the weavers consist of 991 JavaScript lines and 726 lines of Swift code,

[5]Source Lines Of Code

[6]This research was published in [A.1] before this thesis was written. It uses JBoss Drools as a DSL for business rules description instead of a language proposed in Chapter 5. It is available at drools.org.

Table 7.2: SLOC comparison of the conventional approach to the ADD in the UI

Component / SLOC	Conventional	ADD
Web UI	1284 + 83	177
Mobile UI	336	94
Web Layout and widgets	-	197
Mobile Layout and widgets	-	121
Business rules	-	94
Localization	-	256
Web weaver	-	921
Mobile weaver	-	726

which is significant initial overhead comparing to the total SLOC. Furthermore, the ADD also requires reusable server-side inspectors, which expose the concerns through the API, e.g., a data model inspector and a business rules inspector. However, the SLOC of the inspectors is not considered in this case study.

In conclusion, design of the UI of an EIS is a challenge as such a system exposes extensive amount of complex data and allows users maintain them within implemented business processes. The UI has to consider various concerns, which tangle together. The case study presented in this section shows major code and information repetition reduction with the ADD. Every concern identified by the ADD is cleanly separated (Problem 1) and located in the single focal point (Problem 3), while it is automatically reused and distributed over all platforms at runtime (Problem 2). Although the runtime weaving (Problem 4) might seem inefficient, benchmarks show both memory footprint and server CPU use reduction, as UI construction is moved from the server to clients [A.12]. Concerns maintenance no longer depends on size of a system or number of platforms, as there is no manual repetition (Problem 5). Fortunately, while aspect weavers introduce significant additional SLOC, they are reusable and their complexity does not grow with a system scope. Unfortunately, there still remain some manual code but not concern repetition, which is elaborated in detail in [A.1]. Nevertheless, maintenance of such a system requires significantly less efforts comparing to the conventional approach, as modification of each concern means changing only one place.

7.3 Comparison to Alternative Approaches

While proofs of a concept and case studies demonstrating efficiency of the approach provide valuable results, they do not evaluate the approach in a wider context. To reach conclusions about efficiency of the approach, it is necessary to consider existing alternative approaches and their solution for a given challenge. Therefore, to evaluate the

ADD, this section discusses its efficiency in contrast to alternative existing approaches. Unfortunately, as this chapter states in its beginning, it takes major efforts to conduct a valid case study. It requires production-ready implementation of aspect weavers and tools, and access to a real large system to be able to consider real scenarios, and such a task complexity exceeds the scope of this thesis and is left for future work. Therefore, this section evaluates alternative basic approaches as is elaborated in [A.11].

Considering evaluation of the ADD, then representation of the concerns and their manual repetition are essential qualities due to the scope and the focus of the approach. Although, concerns tangling, scattering, and coupling as well as code cohesion can also be taken into account because the ADD focuses on more efficient maintenance of the concerns. Therefore, the conducted case study measures two qualities:

(i) manual repetition of information captured in source code because the ADD focuses on its removal

(ii) cyclomatic complexity in the domain and data layers because the ADD extracts the rules implemented as if-statements and applies them via business contexts

It is essential to remember that the ADD is an abstract high-level approach focused on reduction of manual repetition of the concerns in both horizontal and vertical dimensions. Besides easier maintenance, it aims on overall simplification of source code resulting from untangling of the concerns and their centralized maintenance. In consequence, the case study is designed to measure these claimed qualities.

7.3.1 Approaches to EIS Development

To evaluate the ADD, it is necessary to measure efficiency of the ADD comparing to other existing alternative approaches. Therefore, the case study presented in this section considers four approaches in total, the ADD and three alternatives [A.11].

1. Naive implementation of an EIS without any design patterns or frameworks.
2. Java EE implementation utilizing the specification and attached libraries.
3. Object-based implementation utilizing OOP techniques to reuse concerns.
4. The ADD itself in order to evaluate it.

It seems that the case study considers only naive approaches without advanced techniques, which might threat validity of the case study. However, these are conventional contemporary approaches to development of an EIS. Unfortunately, most existing systems do not utilize any of the advanced approaches to the separation of concerns and fall back to these approaches instead; comparison to alternative and more sophisticated approaches was left for future work, as it requires existence of high quality implementation of the ADD. In conclusion, the validity of results of this case study is limited to naive and conventional approaches to design and development of EISs.

Naive implementation is a reference implementation standing for a worst case scenario. It does not optimize representation or use of the concerns; it captures them in a programming language directly in places of their use, instead. Unfortunately, while this is inefficient, it is common practice in real-world development. In consequence, the concerns are repeated both horizontally and vertically when they are applied.

Java EE implementation is another common approach to EIS development. Contemporary EISs often build on the top of the Java EE or .NET platforms, both with similar underlying principles. While Java does not consider any concerns discussed in this thesis, it provides basic support for business rules. *JSR 303: Bean Validation* [8] uses meta-programming (annotations) as declarative programming to annotate model fields and set additional constraints such as *`@Email`* for strings matching the email pattern or *`@NotBlank`* for non-empty strings. In addition, *JSR 205* [75] enables annotating business operations to restrict access to them and enforce a security policy. For example, *`@RolesAllowed(ADMINISTRATOR)`* makes the operation available only to administrators. Furthermore, RichFaces[7] is a framework for the UI development. It reads a subset of these annotations and automatically reuses the constraints in input fields in UI forms. For example, a string field with *`@Email`* annotation is rendered in a UI form as an input field with client-side email validation.

Unfortunately, the constraints represented via annotations must be invariants, i.e., must be valid at all times, which only works for a few scenarios. Usually, constraints change either with a business context or an execution context but the Java EE is unable to deal with contextual rules. Therefore, it forces developers to tangle these rules into the code as the naive implementation does. However, the approach to express constraints is generic, although its implementation is limited to model fields. To overcome this limitation, for purpose of this case study, a Java EE extension enabling the annotations over methods was introduced. In consequence, it eases declaration of constraints and reduces their scope to business operations, although they are still context-unaware. In conclusion, the implementation enables restriction of both model fields and business operations by constraints. Moreover, domain model invariants are located in a single place, but operation-specific constraints are manually repeated as there is no mechanism to reuse them over multiple methods. Finally, the extension does not affect the presentation layer; it remains limited to original behavior of RichFaces.

Object-oriented implementation uses OOP techniques such as design patterns, encapsulation, and polymorphism to reuse concerns and minimize their manual repetition, as is suggested in [15]. This approach tends to declarative programming similar to was described in the paragraph, and it benefits from its easier extendibility. As this approach utilizes, e.g., *Proxy* and *Visitor* design patterns, it is easier to implement custom validators or group up some common constraints. On the other hand, implementation of validators is less declarative and cannot be inspected through reflection.

[7]http://richfaces.jboss.org

For example, consider an `Issue` from this case study [A.11]. Its fields, such as `name` or `description`, are restricted with length constraints. Every place validating an Issue object invokes these validators to verify the constraints without actually duplicating the validation code. Although it restates the set and configuration of used validators. Therefore, creation of `IssueValidator` encapsulates all validators related to an Issue. Furthermore, context-aware validators, i.e., validators accepting a context, overcome the limitation of Java EE annotations. On the other hand, consider client-side validation in the UI or composition of queries in the data layer. While constraints are the same as in the domain layer, technologies differ. Unfortunately, as the validators are not inspectable, it is not possible to have them automatically vertically transformed into other technologies and languages. In consequence, all validators must be transformed and kept synchronized manually instead.

Aspect-driven implementation follows the ADD approach [A.8] introduced and elaborated in this thesis. Since it perceives concerns in EISs as significantly tangled, scattered, and difficultly maintained, it focuses on their separation, automated transformation, and runtime distribution. It stores them independently in the repository, which introduces a single point of the truth. Significant use of DSLs brings additional benefits into development, such as more efficient maintenance of the concerns and involvement of the domain experts into development. Unfortunately, the approach requires implementation of complex aspect weavers and existence of various DSLs, which significantly increases initial overhead. Fortunately, the tools are reusable among projects, which utilizes the efforts. Empirical studies show that aspect-oriented designs tend to higher stability and lower impact of change requests [49].

Considering the impact of the approach, the most significant benefit is decoupling of the concerns and places of their use, which eliminates repetition of definitions of the concerns (Problem 1). It locates the concerns in the repository and defines composition rules and references into the repository, instead (Problem 3). Then, utilization of the AOP is responsible for runtime context-aware weaving (Problem 4). Furthermore, represented concerns are vertically transformed into various technologies and languages in order to make them reusable in all components, layers, and environments (Problem 2). As a result, the approach claims removal of both horizontal and vertical repetitions of the concerns and saves a significant amount of source code (Problem 5).

7.3.2 Results of the Case Study

To compare the considered approaches and evaluate the ADD in the context of alternative conventional development approaches, consider a small issue tracking system. The case study measures concerns repetition, maintenance efforts, and SLOC[8]. All

[8]Source Lines of Code

Table 7.3: SLOC efficiency of conventional approaches to EIS development

Component	Naive	Java EE	OO	ADD
Domain Model[A]	198	225	198	198
Domain Layer	304	260	283	216
Presentation Layer[B]	207	224	207	207
Presentation Layer[C]	794	771	794	422
Other	0	89[D]	209[E]	273[F]

[A] Including annotations.
[B] Only in Java files, i.e., backing beans.
[C] Only in XML files, i.e., UI description.
[D] Custom annotations.
[E] Validators implementing business rules.
[F] Business rules in a DSL.

four implemented approaches use Java EE 6 platform with JavaServer Faces 2.1 for the UI. Next, Java EE implementation uses Rich Faces for implementation of the UI, which provides partial reuse of model constraints. The ADD implementation uses first generation of a library partially implementing the approach with JBoss Drools 6 as a DSL for business rules description. Unfortunately, the version of the library is neither production-ready nor uses a DSL implementing all the requirements from Chapter 5. However, this proof of the concept implementation is sufficient for the purpose of this case study. In addition, the ADD uses extended implementation of AspectFaces[9], which implements the ADD composition of the UI.

The case study focuses on the domain layer and the presentation layer, as none of the other approaches but the ADD provides any simplification of the data layer. Furthermore, comparing to the original results published in [A.11], these results also consider implementation of the UI using the ADD, which was delivered later after the paper publication. Considering the issue tracking system, the case study identifies 33 business operations restricted by 116 constraints including model invariants, and with 14 different views implemented exposing all 33 business operations.

Table 7.3 shows differences among the approaches in the SLOC metric. While the object-oriented implementation seems almost identical to the naive implementation, the numbers does not show qualitative differences. As the naive implementation restates the concerns such as business rules, the object-oriented implementation encapsulates the concerns in objects (209 lines of code) and repeats only references to these objects. Therefore, although SLOC analysis does not show any significant differences, maintenance of the object-oriented implementation is significantly easier.

The Java EE implementation reduces the SLOC in the domain layer by 14,5 % in trade off increased SLOC in the domain model by 13,5%. It results from movement of business rules from executable code into declarative annotations over the model and business operations. In consequence, while the total SLOC is similar, maintenance

[9]http://aspectfaces.com

Table 7.4: Occurrence of business rules in approaches to EIS development

Constraints	Naive	Java EE	OO	ADD
In model[A]	0	27	0	0
In SQL[B]	38	38	38	0
In business operations[C]	78	3/56[D]	62[E]	0/33[F]
In UI (Java)[G]	0	17	0	0/14[F]
In UI (XML)[H]	183	160	183	0/14[F]
Other	0	12[I]	15[J]	116/33[K]

[A] Annotations above fields and referenced validators.
[B] Constraints in SQL to retrieve data from the database.
[C] Constraints in bodies of business operations.
[D] If conditions/annotations over methods and parameters
[E] Including business rules in non-reusable validators.
[F] Constraints/annotations or tags referencing the context.
[G] Only in Java files, i.e., backing beans.
[H] Only in XML files, i.e., UI description.
[I] Reusable annotations representing constraints.
[J] Reusable validators.
[K] Number of rules/contexts in domain-specific language.

of both approaches is different. The Java EE implementation does not repeat the implementation of the concerns, e.g., does not repeat actual Java conditions in business rules, but uses declarative programming to abstract over it. Consequently, while this approach is slightly less error-prone due to hidden and reused implementation details, the concerns are still manually restated. Similarly, in the UI, Rich Faces reuses only some of model invariants, complex or contextual rules are not supported. Therefore, they must be manually implemented, i.e., repeated. Finally, the SLOC and amount of repetitions is similar, maintenance of source code is significantly different.

The most significant difference is among the aspect-driven implementation and the other implementations. The ADD has the lowest SLOC in all considered components. It indicates the reduction in the domain layer by 17,5 %, 14,6 %, and 13,9 % relative to the naive, Java EE, and object-oriented implementation, respectively. While the absolute values are small as the domain model consists of only four classes, scaling the results to much bigger models, and the reduction is significant. It results from extraction of the concerns in to the separate files and the repository. Table 7.3 shows significant SLOC dedicated specifically to business rules, the other concerns are included in the presentation layer. Furthermore, considering inefficiency of the used Drools language and its significant overhead, the reduction would be even greater having a more efficient language tailored for the ADD.

Unfortunately, the SLOC analysis does not show repetitions and occurrences of the concerns in source code. Therefore, Table 7.4 provides summary of business rules occurrence in the source code. First, it would be very difficult to count all concerns because some of them might be difficult to measure. Second, while the ADD facilitates the concerns separation and automated composition for all the concerns, the other approaches mostly only support business rules. Therefore, counting also other concerns would be inaccurate and strongly advantageous for the ADD. Instead, counting only business rules is accurate and not biased as their representation is optimized by all implementations but the naive.

Consider Table 7.4. The naive implementation is the worst case; there are no efforts devoted to avoiding repetitions. The results show inconvenient distribution of the concern throughout the all layers for all but the aspect-driven implementation. Although, the Java EE implementation with the extension partially deals with business rules, it still leaves many constraints uncovered and tangled into code base, especially in the UI due to limited support of business rules reuse in Rich Faces.

The object-oriented implementation slightly improves the naive implementation as it does not restate the constraint semantics, i.e., how to validate it, anymore. However, it repeats invocation of validators, i.e., constraint references, which is still tedious and error-prone as some occurrences are easy to overlook. Therefore, aggregated validators encapsulate invocation of all partial validators in order to reduce repetitions of the validators. However, it does not impact this case study much as it is too small for this kind of optimization. Furthermore, this optimization of business rules is limited to the domain layer because it is challenging to have the validators automatically transformed into other technology. Therefore, considering validation in the UI, it results in either redefinition of validators or manual repetition of business rules, which is more convenient solution in this case study.

The results confirm that the aspect-driven implementation locates all business rules in the repository as there are no repetitions throughout the system. Furthermore, it repeats neither the invocations nor the rules definitions; it only addresses decoupled business contexts to denote rules to apply. Therefore, considering the table, the contexts are referenced via annotations in Java and tag attributes in XML in JavaServer Faces. Unfortunately, due to limited implementation of the requirements from Chapter 5 in JBoss Drools language, there are significant repetitions in business rules definitions. It results from limited context composition, which is one of the essential requirements for efficient representation of business rules. However, this limitation and all repetitions can be avoided by using a DSL tailored specifically for the ADD. In conclusion, it seems the ADD seems is efficient, although, there is no production-ready implementation to confirm these claims in a large scale case study.

Finally, considering maintenance efforts, the important criteria are 1) high concerns cohesion, 2) low or no manual repetition of concerns, 3) the fewest possible places to edit, and 4) the simplest implementation of business logic without any additional

Table 7.5: McCabe cyclomatic complexity in the domain layer

Metric	Naive	Java EE	OO	ADD
Average complexity	3.18	1.09	1	1
Highest complexity	12	2	1	1
Total complexity	105	36	33	33

context validations. Looking into the results in Table 7.4, the naive implementation is very difficult to maintain. There is high concerns repetition as the code contains 299 business rules while the system domain declares only 116, i.e., increasing to 258 %. Furthermore, business rules are tangled throughout multiple different technologies. This correlates with the method complexity measurement reported in Table 7.5, which shows that every method contains a sequence of if-statements.

The Java EE implementation slightly reduces repetitions, but the number of technologies is increased by Java annotations representing another place to consider. A strong aspect of this approach lies in delegation of the rules semantics to a validation processor scanning the annotations, which reduces error-proneness. Unfortunately, it is quite difficult to implement a custom annotation, which may result in distribution of business rules among annotations and method bodies, instead. This limitation is visible in Table 7.5, which shows that there exist conditions captured inside of methods.

Although the measurement of concern repetitions in the object-oriented implementation suggests no improvement, Table 7.5 confirms a significant reduction of cyclomatic complexity and encapsulation of constraints in singleton validators. All if-statements related to input and context validation are moved into easily testable validator objects. Basically, the idea is similar to Java EE annotations but the implementation is less declarative and simpler to implement. To reduce maintenance efforts, the results of the case study suggest combination the Java EE and object-oriented principles and use of validators, where annotations are too difficult to implement.

Considering the results, the most efficient implementation seems to be the ADD, as it reduces all repetitions, complexity, and the number of technologies with constraint distribution. However, it tangles names of business contexts throughout source code, i.e., references into the repository, to decouple definition and use of the concerns.

In conclusion, although the measurements confirm claimed benefits of the ADD and show it to be the most effortless implementation with easy maintenance, all implementations introduce some overhead. In case of the extended Java EE implementation, it is necessary to design custom annotations and their executors, the object-oriented implementation uses custom validators, and the ADD is a large platform itself with a set of DSLs and aspect weavers. Fortunately, although all these components are complex, they are reusable across projects and the initial efforts can be utilized over multiple projects. However, considering all qualities of the implementations, each of them will perform the best in a different scenario. While the ADD fits large context-aware systems with a complex business domain, a large development team, and various domain experts, the other implementations better fit needs of small and medium projects. Although they do not deliver the top quality source code, overall efforts, i.e., initial overhead plus development and maintenance efforts, are lower comparing to the high initial overhead and complex structure of the ADD implementation.

7.4 Summary

While the aspect-driven development claims its qualities, they must be evaluated in a wider context to confirm these claims, additional qualities, and overall efficiency. Therefore, this chapter presented the most significant results gathered during the research presented in this thesis. While complete and detailed results are elaborated in the papers summarized at the beginning of this chapter, this chapter delivered a proof of the concept demonstrating implementation of the ADD approach into design and development of an UI, compared efficiency of the approach to alternative conventional approaches to software development, which fulfils Objective 3 of this thesis. The results confirm claimed benefits such as efficient separation of concerns, and their automated transformation and context-aware reuse. They show successful removal of manual repetitions from source code as well as eased concerns maintenance focused into the single point of the truth. However, the results also confirm significant initial overhead of the approach and requirement of complex production-ready tools to be able to deploy the ADD into a real production system.

The implementation of the approach into design of an UI demonstrates feasibility of the approach and allows conducting of expertise to evaluate its qualities. The proof of the concept show efficient separation and description of the concerns, while it emphasizes the need for complex tools and languages. It also suggests existence of significant mental barrier as the ADD changes the architecture, code organization, and possibly also the development process.

Quantitative evaluation of the approach shows significant code reduction and efforts comparing to conventional development, which results from automated transformation and reuse of the concerns. However, the case study demonstrates high initial efforts, which significantly reduces efficiency of the approach for small and medium projects. Although the tools such as aspect weavers are reusable across projects.

Finally, the comparison of the aspect-driven implementation to conventional approaches to EIS design and development confirms assumptions make in this summary. The results show overall efficiency of the ADD for large, complex, and long-running projects with a complex business domain, but also demonstrate high initial overhead and requirement of complex tools, which basically make the ADD unusable with small and medium projects. On the other hand, the other approaches are more suitable to small and medium systems, despite being less efficient. The optimization they perform is sufficient and outweighed by lower overhead and easier use.

In conclusion, although the conducted case studies are small, they confirm both claimed benefits and limitations of the ADD, evaluate its qualities, and fulfil Objective 3 of this thesis. The results show efficient separation of concerns and removal of both vertical and horizontal repetitions, which are problems identified in Chapter 1 and addressed by Objectives 1 and 2. Furthermore, while the ADD definition in Chapter 4 assumes the Anemic Domain Model, [A.9] discusses and evaluates deployment of the approach into design of a system with the Rich Domain Model. Next, while the third case study in this chapter only considers conventional approaches to development of an EIS, there also exist advanced and more sophisticated approaches. Unfortunately, conducting of a case study with these approaches puts higher demands on used tools such as DSLs and aspect weavers, which are not ready yet. Therefore, such a case study as well as a large production-size case study are left for future work.

Chapter 8

Conclusion

Conventional approaches to design and development of enterprise information systems reach their limits, since the scale of systems, complexity of business processes, and volume of maintained data continuously grow. Separation, representation, and automated reuse of cross-cutting concerns belong among major challenges in contemporary systems. While the amount of maintained data and complexity of business processes visibly grow with requirements, complexity and amount of business rules, organization of layouts in a user interface, and the number of views in mobile and web applications grow silently since these cross-cutting concerns do not fulfill a prime objective of a system. The concerns cross-cut throughout multiple system layers, components, and technologies, are orthogonal to each other, and compose a multidimensional space, which makes their separation and reuse difficult and challenging. Unfortunately, conventional approaches force developers to manually linearize the multidimensional space, which leads to significant code duplication and information restatement. Subsequently, such code tends to inconsistencies, and its highly error-prone maintenance requires significant efforts. As such, this thesis addressed the separation of concerns in enterprise information systems, and proposed a novel approach to design and development of these systems, which deals with this challenge.

8.1 Contribution of the Thesis

Chapter 1 identified problems arising from limited support of separation of concerns in contemporary implementations. Code tangling, manual information repetition, and absence of central maintenance of concerns in a system result in significant increase of development and maintenance efforts. Considering the usual utilization of various technologies, high error-proneness caused by significant tendency to inconsistencies is a direct consequence of manual information repetition. Consequently, limited support of separation of concerns significantly impacts such a system, degenerates its architecture, and increases the costs of the entire system. This thesis introduced of a novel approach to design and development of enterprise information systems addressing separation of concerns and identified problems, which also emphasizes the role of business rules. In order to evaluate contribution of this thesis, consider the objectives set in Section 1.3:

Objective 1: Define a novel approach addressing separation of concerns

The aspect-driven development approach defined in Chapter 4 introduces a new component into system architecture in order to maintain models of cross-cutting concerns in the single point of the truth (Problem 3). These usually platform-independent models are described independently in domain-specific languages, which removes concerns tangling (Problem 1), removes manual information repetition, and reduces development and maintenance efforts since it enables involvement of domain experts (Problem 5). Then, the concerns are transformed with forward transformation rules into target technologies (Problem 2), and weaved together at runtime, which enables context-aware decisions considering the execution context (Problem 4). Consequently, while the approach is complex, it utilizes multiple existing approaches in order to address the separation of concerns and identified problems.

Objective 2: Propose a mechanism reusing business rules in a system

The aspect-driven development approach recognizes business rules among cross-cutting concerns in Chapter 4, which enables their separation and context-aware reuse by design. Although, complexity of the concern makes its representation challenging. Chapter 5 defined requirements for business rules representation in order to enable their context-aware evaluation (Problem 4), and discussed options of the concern reuse raised from its separation and efficient representation. A domain-specific language may seem to be efficient representation of business rules, suitable for both development and domain experts, however, there exists no language meeting the requirements. And since designing a new language is a challenge itself and requires significant background study and subsequent evaluation, it is left for future work. In consequence, Chapter 5 concluded with a brief example of a language used for evaluation of the approach.

Objective 3: Evaluate the impact of the approach on system design

While the aspect-driven development approach is generic and agnostic to an architecture and a technology, Chapter 6 proposed its implementation into design of the layered architecture, discussed its utilization in distributed environments with native applications for mobile devices, and refined it for the distributed service-oriented architecture. Provided examples demonstrate efficient separation and reuse of concerns, which addresses Problems 1 to 5 and includes reuse of business rules. Chapter 7 presented conducted case studies and proofs of the concept in order to compare the impact of the approach on a system design to conventional development. The results indicate significantly reduced information repetitions (Problem 3), volume of source code, and development and maintenance efforts (Problem 5). Unfortunately, the approach comes with a significant initial barrier. The conduced case study showed that while less efficient approaches are well suited for small projects, large projects significantly benefit from the aspect-driven development and utilize the initial efforts.

In conclusion, this thesis contributes to the state of the art with the aspect-driven development approach, which can be utilized in design of enterprise information systems. The approach addresses identified problems and set objectives of this thesis, and provides an efficient mechanism for separation and context-aware reuse of cross-cutting concerns in trade of a significant initial barrier. While utilization of component-specific aspect weavers reduces efforts, and domain-specific languages are friendly to domain experts, their implementation requires significant investment, which limits efficiency of the approach in small projects. Fortunately, both languages and components are project-agnostic, which makes them reusable and utilizes the efforts.

8.2 Future work

Although the design and evaluation of the approach suggest promising results, there are left some challenges for future work. Design and implementation of a domain-specific language for business rules representation is essential for further evaluation of the approach as well as production-ready implementation of aspect-weavers for common components, architectures, and technologies. Then, it is essential to conduct a large case study and implement the approach into a real system in order to evaluate it in production environment and confirm claimed benefits and limitations.

The separated concerns open wide range of options to their further reuse. Some options are discussed at the end of Chapter 5, for example, reasoning over business rules is beneficial testing activity, which detects cycles or infeasible business contexts. Automated reasoning potentially leads to automated formal verification against a product specification, since the context are transformable into a formal language. Existing research already shows utilization of the concerns in generation of business documentation, which let domain experts review current implementation of a system.

While this thesis discusses utilization of the approach in the service-oriented architecture, its evolved derivations such as Microservices deal with different assumptions and qualities. The future work includes implementation of the approach into other distributed architectures to determine the benefits and evaluate the efficiency.

Finally, while the approach is tailored for enterprise architectures and large information systems, the Internet of Things also deals with large number of controllers, thus suffer from similar challenges as the service-oriented architecture. While implementation of the approach must be efficient, its utilization can detect similar controllers, find inconsistencies, and enable concerns reuse.

Bibliography

[1] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta. Agile Software Development Methods: Review and Analysis. In *Proceedings of ESPOO 2002*. 2002, pages 3–107.

[2] O. Alam, J. Kienzle, and G. Mussbacher. Concern-oriented Software Design. In *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2013, pages 604–621.

[3] S. W. Ambler. *Mapping Objects to Relational Databases*. AmbySoft, 2000.

[4] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, et al. Business Process Execution Language for Web Services, 2003.

[5] M. Baldauf, S. Dustdar, and F. Rosenberg. A Survey on Context-aware Systems. In *International Journal of Ad Hoc and Ubiquitous Computing*. Volume 2. Number 4. Inderscience Publishers. 2007, pages 263–277.

[6] C. Bauer and G. King. *Hibernate in Action*. Manning Publications Co., 2005.

[7] O. Ben-Kiki, C. Evans, and B. Ingerson. Yaml Ain't Markup Language YAML^{TM} Version 1.1. *Technical Report*, 2005.

[8] E. Bernard, S. Peterson, et al. JSR 303: Bean validation. *Java Community Process*, 2009.

[9] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[10] M. Brock et al. MVEL Expression Language. *Available at http://mvel.document-node.com/*, 2001.

[11] P. Browne. *JBoss Drools Business Rules*. Packt Publishing Ltd., 2009.

[12] A. Buckley et al. JSR 175: A Metadata Facility for the Java^{TM} Programming Language. *Java Community Process*, 2004.

[13] E. Burns and R. Kitain. JSR 314: JavaServer Faces 2.0. *Java Community Process*, 2010.

[14] R. P. Buse and W. R. Weimer. Learning a Metric for Code Readability. In *IEEE Transactions on Software Engineering*. IEEE. 2010, pages 546–558.

[15] T. Cerny and M. J. Donahoo. How to Reduce Costs of Business Logic Maintenance. In *Computer Science and Automation Engineering (CSAE)*. Volume 1. IEEE. 2011, pages 77–82.

[16] T. Cerny and M. J. Donahoo. On Separation of Platform-Independent Particles in User Interfaces. In *Cluster Computing*. Springer. 2015, pages 1215–1228.

[17] T. Cerny, M. J. Donahoo, and E. Song. Towards Effective Adaptive User Interfaces Design. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*. ACM. 2013, pages 373–380.

[18] T. Cerny, M. Macik, M. J. Donahoo, and J. Janousek. On Distributed Concern Delivery in User Interface Design. In *Computer Science and Information Systems*. Volume 12. Number 2. 2015, pages 655–681.

[19] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. *W3C Recommendation*, 26, 2007.

[20] M. Chinosi and A. Trombetta. BPMN: An Introduction to the Standard. In *Computer Standards & Interfaces*. Volume 34. Number 1. Elsevier. 2012, pages 124–134.

[21] I. Choi. A Study on Rule Separation Based on AOP for an Efficient Service System. In *Pacific Science Review A: Natural Science and Engineering*. Volume 17. Number 2. Elsevier. 2015, pages 51–60.

[22] Y. K. Choi, J. S. Yang, and J. Jeong. Application Framework for Multi Platform Mobile Application Software Development. In *Advanced Communication Technology, 2009. ICACT 2009. 11th International Conference On*. Volume 1. IEEE. 2009, pages 208–213.

[23] K. Chung et al. JSR 245: JavaServer$^{\text{TM}}$ Pages. *Java Community Process*, 2006.

[24] M. A. Cibrán, M. D'hondt, and V. Jonckers. Aspect-Oriented Programming for Connecting Business Rules. In *Proceedings of the 6th International Conference on Business Information Systems*. Volume 6. Number 7. 2003.

[25] K. Czarnecki, U. W. Eisenecker, and K. Czarnecki. *Generative Programming: Methods, Tools, and Applications*, volume 16. Addison-Wesley, 2000.

[26] L. Da Xu. Enterprise Systems: State of the Art and Future Trends. In *IEEE Transactions on Industrial Informatics*. IEEE. 2011, pages 630–640.

[27] L. DeMichiel. JSR 317: Java$^{\text{TM}}$ Persistence API, version 2.0. *Java Community Process*, 2009.

[28] L. DeMichiel, W. Shannon, et al. JSR 366: Java$^{\text{TM}}$ Platform, Enterprise Edition 8 (Java EE 8) Specification. *Java Community Process*, 2017.

[29] B. Demuth, H. Hußmann, and S. Loecher. OCL as a Specification Language for Business Rules in Database Applications. In *International Conference on the Unified Modeling Language*. Springer. 2001, pages 104–117.

[30] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, et al. Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering*. Springer. 2017, pages 195–216.

[31] W. J. Dzidek, E. Arisholm, and L. C. Briand. A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance. In *IEEE Transactions on Software Engineering*. Volume 34. IEEE. 2008, pages 407–432.

[32] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, et al. Do Crosscutting Concerns Cause Defects? In *IEEE Transactions on Software Engineering*. Volume 34. Number 4. IEEE. 2008, pages 497–515.

[33] M. Endrei, J. Ang, A. Arsanjani, S. Chua, et al. *Patterns: Service-Oriented Architecture and Web Services*. IBM Corporation, International Technical Support Organization, 2004.

[34] H. Ferentschik, G. Morling, and G. Smet. *Hibernate Validator 6 - JSR 380 Reference Implementation: Reference Guide*. Red Hat Middleware, 2018.

[35] J. I. Fernández Villamor, C. A. Iglesias Fernandez, and M. Garijo Ayestaran. Microservices: Lightweight Service Descriptions for REST Architectural Style. In *Proceedings of the International Conference on Agents and Artificial Intelligence*. INSTICC, Institute for Systems, Technologies of Information, Control, and Communication. 2010.

[36] R. T. Fielding. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[37] R. Filman, T. Elrad, S. Clarke, and M. Akşit. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.

[38] C. L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. In *Readings in Artificial Intelligence and Databases*. Elsevier. 1988, pages 547–559.

[39] A. Forward and T. C. Lethbridge. The Relevance of Software Documentation, Tools and Technologies: A Survey. In *Proceedings of the 2002 ACM Symposium on Document Engineering*. ACM. 2002, pages 26–33.

[40] M. Fowler. Anemic Domain Model. *Available at https://martinfowler.com/bliki/AnemicDomainModel.html*, 2003. [Accessed on January 12, 2019].

[41] M. Fowler. *Domain-Specific Languages*. Addison-Wesley, 2010.

[42] M. Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, 2002.

[43] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[44] M. Fowler and J. Lewis. Microservices. *Available at https://martinfowler.com/articles/microservices.html*, 2014. [Accessed on January 12, 2019].

[45] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[46] D. Garlan and M. Shaw. An Introduction to Software Architecture. In *Advances in Software Engineering and Knowledge Engineering.* World Scientific. 1993, pages 1–39.

[47] D. Ghosh. *DSLs in Action.* Manning Publications Co., 2010.

[48] R. E. Giachetti. *Design of Enterprise Systems: Theory, Architecture, and Methods.* CRC Press, 2016.

[49] P. Greenwood, T. Bartolomei, E. Figueiredo, et al. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In *European Conference on Object-Oriented Programming.* Springer. 2007, pages 176–200.

[50] M. J. Hadley. Web Application Description Language (WADL), 2006.

[51] D. Hay, K. A. Healy, J. Hall, et al. Defining Business Rules - What Are They Really?, 2000.

[52] F. Hayes-Roth. Rule-Based Systems. In *Communications of the ACM.* Volume 28. Number 9. ACM. 1985, pages 921–932.

[53] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, et al. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. In *W3C Member submission.* Volume 21. 2004.

[54] P. Hudak. Domain-Specific Languages. In *Handbook of Programming Languages.* Volume 3. Number 39-60. 1997.

[55] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master.* Addison-Wesley, 2000.

[56] I. Jacobson, G. Booch, J. Rumbaugh, J. Rumbaugh, et al. *The Unified Software Development Process*, volume 1. Addison-Wesley, 1999.

[57] S. K. Johnson and A. W. Brown. A Model-Driven Development Approach to Creating Service-Oriented Solutions. In *International Conference on Service-Oriented Computing.* Springer. 2006, pages 624–636.

[58] S. Kelly and J. P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, 2008.

[59] R. Kennard, E. Edmonds, and J. Leaney. Separation Anxiety: Stresses of Developing a Modern Day Separable User Interface. In *Human System Interactions. HSI'09. 2nd Conference on*. IEEE. 2009, pages 228–235.

[60] R. Kennard and J. Leaney. Towards a General Purpose Architecture for UI Generation. In *Journal of Systems and Software*. Volume 83. Number 10. Elsevier. 2010, pages 1896–1906.

[61] R. Kennard and R. Steele. Application of Software Mining to Automatic User Interface Generation. In *International Conference on Software Methods and Tools*. IOS Press. 2008.

[62] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, et al. *Aspect-Oriented Programming*. Springer, 1997.

[63] A. G. Kleppe, J. B. Warmer, and W. Bast. *MDA Explained, The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[64] G. Knolmayer, R. Endl, and M. Pfahrer. Modeling Processes and Workflows by Business Rules. In *Business Process Management*. Springer. 2000, pages 16–29.

[65] V. Kulkarni and S. Reddy. Separation of Concerns in Model-Driven Development. In *IEEE Software*. Volume 20. Number 5. IEEE. 2003, pages 64–69.

[66] R. Laddad. *AspectJ in Action: Enterprise AOP with Spring Applications*. Manning Publications Co., 2009.

[67] C. Larman. *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Iterative Development*. Prentice Hall, 2005.

[68] S. Leblanc, G. Mussbacher, J. Kienzle, and D. Amyot. Narrowing the Gaps in Concern-Driven Development. In *Model-Driven Requirements Engineering Workshop (MoDRE), 2012 IEEE*. IEEE. 2012, pages 19–28.

[69] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice Hall, 1994.

[70] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.

[71] E. Al-Masri and Q. H. Mahmoud. Discovering the Best Web Service. In *Proceedings of the 16th International Conference on World Wide Web*. ACM. 2007, pages 1257–1258.

[72] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. In *ACM Computing Surveys (CSUR)*. Volume 37. Number 4. ACM. 2005, pages 316–344.

[73] A. Mesbah and A. Van Deursen. Crosscutting Concerns in J2EE Applications. In *Proceedings of the Seventh IEEE International Symposium on Web Site Evolution*. IEEE. 2005, pages 14–21.

[74] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[75] R. Mordani et al. JSR 250: Common Annotations for the Java^{TM} Platform. *Java Community Process*, 2006.

[76] A. Moreira, A. Rashid, and J. Araujo. Multi-Dimensional Separation of Concerns in Requirements Engineering. In *Proceedings oF 13th IEEE International Conference on Requirements Engineering (RE'05)*. IEEE. 2005, pages 285–296.

[77] T. Morgan. *Business Rules and Information Systems: Aligning IT with Business Goals*. Addison-Wesley, 2002.

[78] B. Morin, O. Barais, J. M. Jezequel, F. Fleurey, et al. Models at Runtime to Support Dynamic Adaptation. In *Computer*. IEEE. 2009, pages 44–51.

[79] G. Morling. JSR 380: Bean Validation 2.0. *Java Community Process*, 2017.

[80] S. Muthanna, K. Kontogiannis, K. Ponnambalam, and B. Stacey. A Maintainability Model for Industrial Software Systems Using Design Level Metrics. In *Proceedings Seventh Working Conference on Reverse Engineering*. IEEE. 2000, pages 248–256.

[81] L. Nemuraite, L. Ceponiene, and G. Vedrickas. Representation of Business Rules in UML&OCL Models for Developing Information Systems. In *IFIP Working Conference on The Practice of Enterprise Modeling*. Springer. 2008, pages 182–196.

[82] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, 2008.

[83] H. Ossher and P. Tarr. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. In *Software Architectures and Component Technology*. Springer, 2002, pages 293–323.

[84] M. P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering*. IEEE. 2003, pages 3–12.

[85] S. Penchikala. Domain Driven Design and Development in Practice. *Available at https://www.infoq.com/articles/ddd-in-practice*, 2008. [Accessed on January 12, 2019].

[86] R. Perrey and M. Lycett. Service-Oriented Architecture. In *Proceedings of the 2003 Symposium on Applications and the Internet Workshops (SAINT'03 Workshops)*. IEEE. 2003, pages 116–119.

[87] M. Potel. *MVP: Model-View-Presenter the Taligent Programming Model for C++ and Java.* Taligent Inc, 1996.

[88] W. Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, 1994.

[89] E. Putrycz and A. W. Kark. Connecting Legacy Code, Business Rules and Documentation. In *Rule Representation, Interchange and Reasoning on the Web.* Springer, 2008, pages 17–30.

[90] A. Rahmani, V. Rafe, S. Sedighian, et al. An MDA-Based Modeling and Design of Service Oriented Architecture. In *Proceedings of the 6th International Conference on Computational Science.* Springer. 2006, pages 578–585.

[91] J. Rao and X. Su. A Survey of Automated Web Service Composition Methods. In *International Workshop on Semantic Web Services and Web Process Composition.* Springer. 2004, pages 43–54.

[92] M. P. Robillard and G. C. Murphy. Representing Concerns in Source Code. In *ACM Transactions on Software Engineering and Methodology (TOSEM).* Volume 16. Number 1. ACM. 2007. DOI: 10.1145/1189748.1189751.

[93] F. Rosenberg and S. Dustdar. Business Rules Integration in BPEL - A Service-Oriented Approach. In *Proceedings of the Seventh IEEE International Conference on E-Commerce Technology.* IEEE. 2005, pages 476–479.

[94] R. G. Ross. *The Business Rule Book: Classifying, Defining and Modeling Rules.* Business Rule Solutions, 1997.

[95] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, et al. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[96] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 2004.

[97] M. Salatino, M. De Maio, and E. Aliverti. *Mastering JBoss Drools 6.* Packt Publishing Ltd., 2016.

[98] D. C. Schmidt. Model-Driven Engineering. In *Computer.* IEEE. 2006.

[99] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. In *IEEE Software.* Volume 20. Number 5. IEEE. 2003, pages 42–45.

[100] J. Shao and C. Pound. Extracting Business Rules from Information Systems. In *BT Technology Journal.* Volume 17. Springer. 1999, pages 179–186.

[101] A. Sill. The Design and Architecture of Microservices. In *IEEE Cloud Computing.* Volume 3. Number 5. IEEE. 2016, pages 76–80.

[102] S. Smith et al. Common Web Application Architectures. *Available at https://docs.microsoft.com/en-us/dotnet/standard/modern-web-apps-azure-architecture/common-web-application-architectures*, 2018. [Accessed on January 12, 2019].

[103] D. Spiewak and T. Zhao. ScalaQL: Language-Integrated Database Queries for Scala. In *International Conference on Software Language Engineering.* Springer. 2009, pages 154–163.

[104] M. Stoerzer and S. Hanenberg. A Classification of Pointcut Language Constructs. In *Workshop on Software-Engineering Properties of Languages and Aspect Technologies.* ACM. 2005.

[105] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering.* ACM. 1999, pages 107–119.

[106] B. Theodoulidis and A. Youdeowei. Business rules: Towards Effective Information Systems Development. In *Business Information Systems - Uncertain Futures.* 2000, pages 313–321.

[107] M. Thorpe and C. Ke. Simple Rule Markup Language (SRML): A General XML Rule Representation for Forward-Chaining Rules. In *XML Coverpages.* 2001.

[108] J. Vanderdonckt. A MDA-Compliant Environment for Developing User Interfaces of Information Systems. In *International Conference on Advanced Information Systems Engineering.* Springer. 2005, pages 16–31.

[109] M. Villamizar, O. Garcés, H. Castro, M. Verano, et al. Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud. In *Proceedings of the 10th Computing Colombian Conference.* IEEE. 2015, pages 583–590.

[110] M. Voelter, S. Benz, C. Dietrich, et al. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages.* Dslbook.org, 2013.

[111] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. Towards User-Friendly Projectional Editors. In *International Conference on Software Language Engineering.* Springer. 2014, pages 41–61.

[112] M. Wand, G. Kiczales, and C. Dutchyn. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. In *ACM Transactions on Programming Languages and Systems (TOPLAS).* ACM. 2004, pages 890–910.

[113] X. Wang, J. Sun, X. Yang, S. Maddineni, et al. Business Rules Extraction from Large Legacy Systems. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering.* IEEE. 2004, pages 249–258.

[114] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2003.

[115] A. I. Wasserman. Software Engineering Issues for Mobile Application Development. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research.* ACM. 2010, pages 397–400.

[116] M. Weske. Business Process Management Architectures. In *Business Process Management.* Springer. 2012, pages 333–371.

[117] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. C. Hong, et al. Best Practices for Scientific Computing. In *PLoS Biology.* Volume 12. Number 1. Public Library of Science. 2014, e1001745. DOI: 10.1371/journal.pbio.1001745.

[118] M. Zur Muehlen and M. Indulska. Modeling Languages for Business Processes and Business Rules: A Representational Analysis. In *Information Systems.* Number 4. Elsevier. 2010, pages 379–390.

List of Publications

Journals with Impact Factor

[A.1] K. Cemus (65 %), F. Klimes (15 %), O. Kratochvil (10 %), and T. Cerny (10 %). Separation of Concerns for Distributed Cross-platform Context-aware User Interfaces. In *Cluster Computing*. Springer. 2017, pages 2355–2362. DOI: 10.1007/s10586-017-0794-7. (WoS, Scopus, IF=1.601 (Q2))

Other Refereed Journals

[A.2] K. Cemus (90 %) and T. Cerny (10 %). Automated Extraction of Business Documentation in Enterprise Information Systems. In *ACM SIGAPP Applied Computing Review*. Volume 16. Number 4. ACM. 2017, pages 5–13. DOI: 10.1145/3040575.3040576. (WoS)

Cited in: [B.6]

[A.3] T. Cerny (25 %), K. Cemus (25 %), M. J. Donahoo (25 %), and E. Song (25 %). Aspect-driven, Data-reflective and Context-aware User Interfaces Design. In *ACM SIGAPP Applied Computing Review*. Volume 13. Number 4. ACM. 2013, pages 53–66. DOI: 10.1145/2577554.2577561.

Cited in: [B.1, B.12, B.10, B.9, B.2] and 4 more

Conference Papers Excerpted in Web of Science

[A.4] K. Cemus (60 %), F. Klimes (30 %), and T. Cerny (10 %). Aspect-driven Context-aware Services. In *Proceedings of the Federated Conference on Computer Science and Information Systems*. Volume 11. IEEE. 2017, pages 1307–1314. ISBN: 978-8-3946-2537-5. DOI: 10.15439/2017F397. (WoS, Scopus)

[A.5] K. Cemus (50 %), F. Klimes (25 %), O. Kratochvil (15 %), and T. Cerny (10 %). Distributed Multi-platform Context-aware User Interface for Information Systems. In *Proceedings of the 6th International Conference on IT Convergence and Security (ICITCS 2016).* IEEE. 2016, pages 172–175. ISBN: 978-1-5090-3765-0. DOI: 10.1109/ICITCS.2016.7740327. (WoS, Scopus)

[A.6] K. Cemus (90 %) and T. Cerny (10 %). Business Documentation Derivation from Aspect-driven Enterprise Information Systems. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems.* 2016, pages 153–158. ISBN: 978-1-4503-4455-5. DOI: 10.1145/2987386.2987402. (WoS, Scopus)

[A.7] K. Cemus (85 %), T. Cerny (5 %), and M. J. Donahoo (10 %). Automated Business Rules Transformation into a Persistence Layer. In *Procedia Computer Science.* Volume 62. Elsevier. 2015, pages 312–318. DOI: 10.1016/j.procs.2015.08.391. (WoS, Scopus)

Cited in: [B.5, B.8, B.3]

[A.8] K. Cemus (75 %) and T. Cerny (25 %). Aspect-driven Design of Information Systems. In *International Conference on Current Trends in Theory and Practice of Informatics.* Springer. 2014, pages 174–186. ISBN: 978-3-319-04297-8. DOI: 10.1007/978-3-319-04298-5_16. (WoS, Scopus, CORE B)

Cited in: [B.4, B.9]

Conference Papers Excerpted in Scopus

[A.9] K. Cemus (60 %), T. Cerny (20 %), L. Matl (10 %), and M. J. Donahoo (10 %). Aspect, Rich, and Anemic Domain Models in Enterprise Information Systems. In *International Conference on Current Trends in Theory and Practice of Informatics.* Springer. 2016, pages 445–456. ISBN: 978-3-662-49191-1. DOI: 10.1007/978-3-662-49192-8_36. (Scopus, CORE B)

Cited in: [B.7]

[A.10] K. Cemus (75 %), T. Cerny (15 %), L. Matl (5 %), and M. J. Donahoo (5 %). Enterprise Information Systems: Comparison of Aspect-driven and MVC-like Approaches. In *Proceedings of the 2015 Conference on Research in Adaptive and Convergent Systems.* ACM. 2015, pages 330–336. ISBN: 978-1-4503-3738-0. DOI: 10.1145/2811411.2811477. (Scopus)

Cited in: [B.11]

[A.11] K. Cemus (80 %), T. Cerny (15 %), and M. J. Donahoo (5 %). Evaluation of Approaches to Business Rules Maintenance in Enterprise Information Systems. In *Proceedings of the 2015 Conference on Research in Adaptive and Convergent Systems*. ACM. 2015, pages 324–329. ISBN: 978-1-4503-3738-0. DOI: 10.1145/2811411.2811476. (Scopus)

[A.12] T. Cerny (30 %), L. Matl (25 %), K. Cemus (25 %), and M. J. Donahoo (20 %). Evaluation of Separated Concerns in Web-based Delivery of User Interfaces. In *Lecture Notes in Electrical Engineering - Information Science and Applications*. Volume 339. Springer. 2015, pages 933–940. DOI: 10.1007/978-3-662-46578-3_111. (Scopus)

Other Refereed Conference Papers

[A.13] K. Cemus (100 %). Context-aware Input Validation in Information Systems. In *POSTER 2016 - 20th International Student Conference on Electrical Engineering*. Czech Technical University. 2016, pages 1–5. ISBN: 978-80-01-05950-0.

[A.14] K. Cemus (100 %). Evaluation of Business Rules Maintenance in Enterprise Information Systems. In *POSTER 2015 - 19th International Student Conference on Electrical Engineering*. Czech Technical University. 2015, pages 1–5. ISBN: 978-80-01-05499-4.

[A.15] K. Cemus (95 %) and T. Cerny (5 %). Towards Effective Business Logic Design. In *POSTER 2013 - 17th International Student Conference on Electrical Engineering*. Czech Technical University. 2013. ISBN: 978-80-01-05242-6.

Citations

Journals with Impact Factor and Patents

[B.1] A. Abdyssalam Alhaag, G. Savic, G. Milosavljevic, M. Tima Segedinac, et al. Executable Platform for Managing Customizable Metadata of Educational Resources. In *The Electronic Library.* Volume 36. Number 6. Emerald Publishing Limited. 2018, pages 962–978. DOI: 10.1108/EL-04-2017-0079. (WoS, Scopus, IF=0.8 (Q3))

[B.2] S. Yu and C. Chandra. UI-driven Model Extensibility in Multi-tier Applications. In US Patent App. 10/073,604. Google Patents. 2018. (Patent)

[B.3] G. Villarrubia, J. F. D. Paz, D. H. Iglesia, and J. Bajo. Combining Multi-agent Systems and Wireless Sensor Networks for Monitoring Crop Irrigation. In *Sensors.* Volume 17. Number 8. Multidisciplinary Digital Publishing Institute. 2017, page 1775. DOI: 10.3390/s17081775. (WoS, Scopus, IF=2.475 (Q2))

[B.4] K. Santhi, G. Zayaraz, and V. Vijayalakshmi. Resolving Aspect Dependencies for Composition of Aspects. In *Arabian Journal for Science and Engineering.* Volume 40. Number 2. Springer. 2015, pages 475–486. DOI: 10.1007/s13369-014-1454-3. (WoS, Scopus, IF=1.092 (Q3))

Other Citations

[B.5] B. Hnatkowska and T. Gawęda. Automatic Processing of Dynamic Business Rules Written in a Controlled Natural Language. In *Towards a Synergistic Combination of Research and Practice in Software Engineering.* Springer. 2018, pages 91–103. (Scopus)

[B.6] M. Muji and D. Bică. Qualitative Metrics For Development Technologies Evaluation In Database-driven Information Systems. In *Scientific Bulletin of the Petru Maior University of Targu Mures.* Volume 15. Number 2. 2018.

[B.7] Z. Mushtaq, G. Rasool, and B. Shahzad. Detection of J2EE Patterns Based on Customizable Features. In *International Journal of Advanced Computer Science and Applications*. Volume 8. Number 1. 2017, pages 361–376. (WoS)

[B.8] S. Rodriguez, J. Camilo, et al. Automatización de Pruebas de Software Web Basada en Reglas de Negocio, 2017.

[B.9] D. Weber. *A Constraint-Based Approach to Data Quality in Information Systems*. PhD thesis, ETH Zurich, 2017.

[B.10] I. Paliokas, S. Segkouli, D. Tzovaras, and C. Karagiannidis. A Dynamic Interface Adaptation Approach for Accessible Immersive Environments. In *10th International Conference on Interfaces and Human Computer Interaction, Multiconference on Computer Science and Information Systems (MCCSIS 2016)*. 2015. (Scopus)

[B.11] J. E. Davis. *Temporal Meta-model Framework for Enterprise Information Systems (EIS) Development*. PhD thesis, Curtin University, 2014.

[B.12] P. P. Dey, B. R. Sinha, G. W. Romney, M. Amin, et al. Innovative User Interface Engineering. In *International Conference on Innovative Engineering Technologies*. 2014.

Appendix A

List of Abbreviations

ADD	Aspect-driven Development
ADDA	Aspect-driven Development Approach
ADM	Anemic Domain Model
AI	Artificial Intelligence
AOP	Aspect-oriented Programming
API	Application Programming Interface
AST	Abstract Syntax Tree
BPEL	Business Process Execution Language
BPMN	Business Process Model And Notation
CASE	Computer-aided Software Engineering
CDD	Concern-driven Development
CPU	Central Processing Unit
DRY	Don't Repeat Yourself
DSL	Domain-specific Language
DSM	Domain-specific Modeling
EIS	Enterprise Information System
EL	Expression Language
GP	Generative Programming
GPL	General-purpose Language
GRASP	General Responsibility Assignment Software Patterns
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IP	Internet Protocol
IT	Information Technology
Java EE	Java Enterprise Edition
JSON	Javascript Object Notation
JSR	Java Specification Request

KISS	Keep It Simple, Stupid Principle
MDD	Model-driven Development
MPS	Meta-programming System
OCL	Object-contraint Language
OO	Object-oriented
OOP	Object-oriented Programming
ORM	Object-relational Mapping
POJO	Plain Old Java Object
RDM	Rich Domain Model
READ	Rich Entity Aspect/audit Design
SLOC	Source Lines Of Code
SOA	Service-oriented Architecture
SOAP	Simple Object Access Protocol
SOLID	Set of Object-oriented Principles
SQL	Structured Query Language
UC	Use Case
UI	User Interface
UML	Unified Modeling Language
VAT	Value-added Tax
WADL	Web Application Description Language
WSDL	Web Services Description Language
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations