

Article

# Data Integration and Interoperability: Towards a Model-Driven and Pattern-Oriented Approach

Roland J. Petrasch <sup>1,\*</sup> and Richard R. Petrasch <sup>2</sup><sup>1</sup> Department of Informatics and Media, Berliner Hochschule für Technik, 13353 Berlin, Germany<sup>2</sup> NIVA American International School, Bangkok 10240, Thailand; 1944@niva.ac.th

\* Correspondence: roland.petrasch@bht-berlin.de; Tel.: +66-092-6104285

**Abstract:** Data integration is one of the core responsibilities of EDM (enterprise data management) and interoperability. It is essential for almost every digitalization project, e.g., during the migration from a legacy ERP (enterprise resource planning) software to a new system. One challenge is the incompatibility of data models, i.e., different software systems use specific or proprietary terminology, data structures, data formats, and semantics. Data need to be interchanged between software systems, and often complex data conversions or transformations are necessary. This paper presents an approach that allows software engineers or data experts to use models and patterns in order to specify data integration: it is based on data models such as ER (entity-relationship) diagrams or UML (unified modeling language) class models that are well-accepted and widely used in practice. Predefined data integration patterns are combined (applied) on the model level leading to formal, precise, and concise definitions of data transformations and conversions. Data integration definitions can then be executed (via code generation) so that a manual implementation is not necessary. The advantages are that existing data models can be reused, standardized data integration patterns lead to fast results, and data integration specifications are executable and can be easily maintained and extended. An example transformation of elements of a relational data model to object-oriented data structures shows the approach in practice. Its focus is on data mappings and relationships.

**Keywords:** enterprise data management; enterprise interoperability; data integration; data integration pattern; DIP; UML; UML profile; enterprise integration pattern; EIP; model mapping; model transformation; data model; relational data model; object-oriented data model



**Citation:** Petrasch, R.J.; Petrasch, R.R. Data Integration and Interoperability: Towards a Model-Driven and Pattern-Oriented Approach. *Modelling* **2022**, *3*, 105–126. <https://doi.org/10.3390/modelling3010008>

Academic Editors: Greg Zacharewicz, Nicolas Daclin, Guy Doumeings and Hezam Haidar

Received: 16 November 2021

Accepted: 22 February 2022

Published: 27 February 2022

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Copyright:** © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction and Methodology

### 1.1. Introduction to Data Integration

Enterprise software systems such as ERP, CRM (customer relationship management) and eCommerce, in addition to IoT (Internet of Things) and EDI (electronic data interchange) applications deal with simple or complex data, either in the form of operational data in the context of OLTP (online transaction processing) or analytical data during OLAP (online analytical processing) [1]. Therefore, data elements organized in different and often incompatible data models need to be integrated, i.e., consolidated, converted, merged, or transferred, so that a consistent and up-to-date data representation exists. A data warehouse for analytics or a relational database management system (RDBMS) that serves as an operational persistence layer and data hub for external eCommerce systems are examples of software applications that need data integration. To make a long story short: when it comes to IT, data integration is (necessary) almost everywhere.

Data integration “involves combining data from several disparate sources, which are stored using various technologies and provide a unified view of the data” [2] and is an aspect of enterprise interoperability (EI) that “refers to the ability of interactions between enterprise systems” [3]. In the context of the framework for enterprise interoperability [4],

the focus is on the data (dimension “interoperability concern”) and partly on the interoperability categories “conceptual” and “technological”. In other words, whenever two software systems or services exchange data, data integration is needed. The complexity of data integration depends on various factors such as data models, data formats, and data precisions; however, in most cases it is non-trivial, so that a systematic and well-defined approach is necessary. This, and the fact that in probably every digitalization project in practice, data exchange and integration is an important topic that has led to the development of a data integration pattern (DIP) catalog and the model-driven and pattern-oriented approach for data integration that is described here.

Before the complete methodology is explained and exemplified in Section 4, an overview of the methodology is provided in the Section 1.2. followed by a closer look at existing scientific and practical approaches for data integration (Section 2) and a detailed explanation of data integration and DIPs (Section 3).

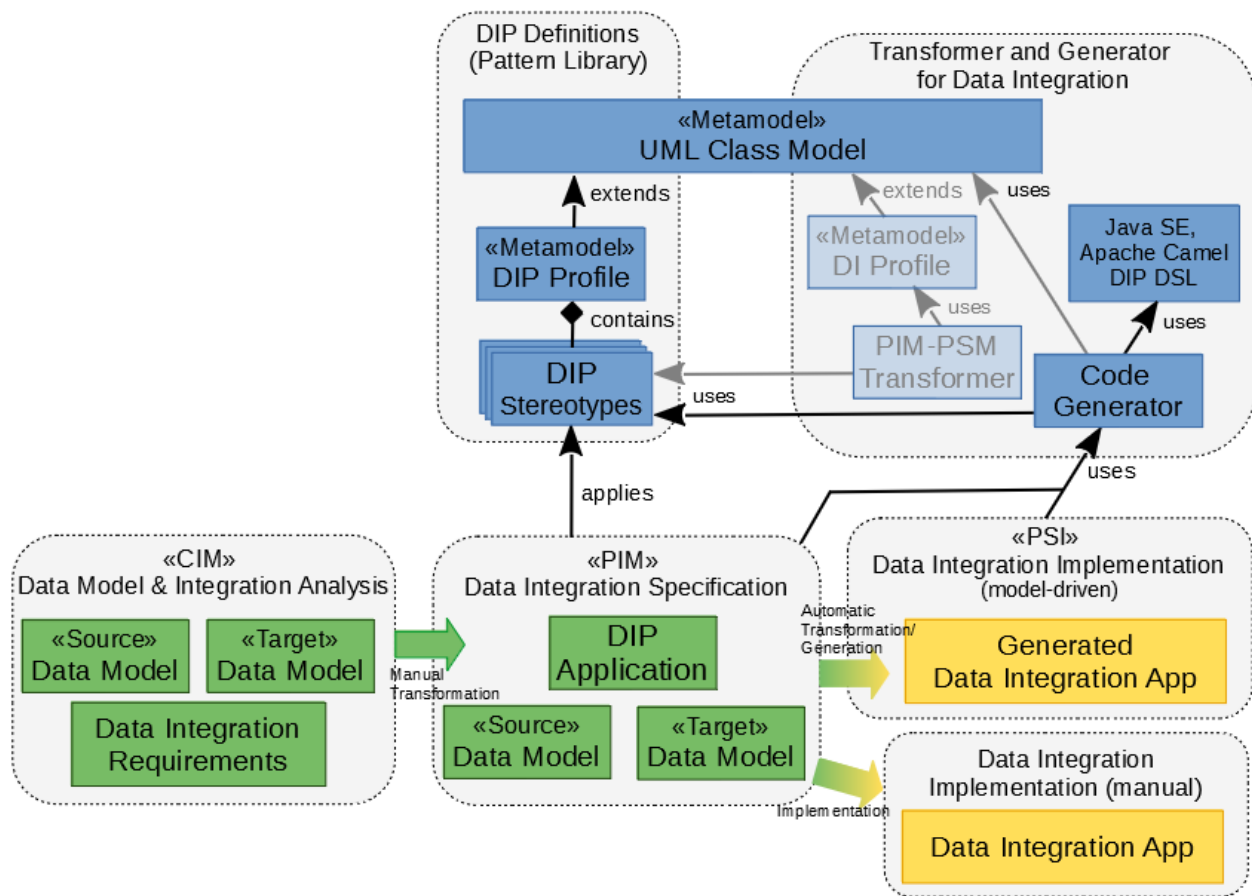
### 1.2. Methodology for a Model-Driven and Pattern-Oriented Data Integration: Overview

Some data integration tasks such as “mapping of relational data (tables) to object-oriented classes” occur quite often and follow certain patterns that have been identified and described as DIPs [5]. DIPs can be used during software development and system integration projects in the same way other pattern libraries are used, for instance, software design patterns or data modeling patterns. DIPs can also be combined with a model-driven methodology leading to formal, transformable, and executable data transformation specifications. This is possible because data integration tasks can be specified and modeled like almost any other part of a software system. OMG’s MDA (model-driven architecture, [6,7]) and the UML (unified modeling language, [8]) combined with the profile mechanism for the definition of a lightweight metamodel for DIPs, are used for this purpose.

Figure 1 gives an overview of the approach, which consists of three main parts: (1) the definition of a DIP library and the development of a code generator (see the upper area with blue symbols), (2) the data integration analysis and specification using DIPs (in the context of a practical project), and (3) the development of a data integration software system using either a generated application or a manual implementation. The details for the first part of the approach are as follows:

- DIP definitions: Formal definitions of DIPs are an integral and essential part of the approach. The outcome is a DIP catalog (or library) that is provided to the system analyst or data expert. As mentioned before, UML class diagrams are used for the DIP definitions using a UML DIP profile that extends the meta-classes “Class” and defines stereotypes with properties (tagged values). This lightweight approach for metamodeling is useful in cases such as DIPs where existing UML meta elements can be reused and extended, avoiding the introduction of new metamodel elements on M2 (metamodel level). DIPs are used in the context of PIMs (platform independent model) [6] (p. 11);
- Code generation for data integrations: This part represents the development of a transformer and code generator that can be used in data integration projects. The transformation of a PIM requires a DI profile that leads to a PSM (platform specific model) [6] (p. 11). The transformer is omitted here in order to simplify the methodology and keep the example as compact as possible. Therefore, a code generator was developed that creates code as a PSI (platform specific implementation (PSI is an officially introduced concept in the MDA Guide but since the “PIM/PSM pattern relates these different levels of abstraction and the concepts of PIM and PSM are, therefore, relative” [6] (p. 11), the last PIM–PSM transformation with text and code as its output artefacts leads to the term PSI (instead of PSM) [9] (p. 456), see also [10] (p. 3.)) directly from PIMs. This direct PIM–PSI approach lacks the flexibility that comes with PIM–PSM transformations but is considered sufficient for a PoC (proof of concept). The code generator for M2T (model-to-text transformation) uses a template

engine (Eclipse Obeo Aceleo, [11]) that complies with OMG's Mof2Text (MOF to text template language, [10]).



**Figure 1.** Model-driven and pattern-oriented methodology for data integration: overview.

The application and usage of DIPs and the code generator take place in practical projects consisting of the following analysis, specification, and implementation tasks:

- Data model and integration analysis: during this mainly manual step that produces a CIM (computation independent model) [6] (p. 8), source and target data models and elements are analyzed and requirements of data integration are defined and documented;
- Data integration specification: During the DI specification, DIPs are applied to the source and target data elements leading to DI-specific UML class diagrams that act as a PIM (M1, model level). The application of stereotypes to UML model elements on M1 is often called “model marking” (or “to mark up a model” [12]). Class models marked with DIPs can then be used for further transformations or code generations because of their formal characteristic. The applied DIPs map source and target data elements, define conversions, aggregations, and other transformation logic;
- Data integration implementation: In order to be able to generate an executable data integration application that connects two or more systems providing source and target data, the DI specification that represents the PIM (or PSM) needs to be provided. The generator performs a model-to-text transformation (in terms of OMG's MDA) where the text is the code and other artifacts represent the PSI on M1 (model level). As an alternative, the implementation can be performed manually without a code generator. Again, the output is an executable application for (reoccurring) data integration tasks. In both scenarios, a Java DSL (domain specific language) for DIPs developed to support DI implementation projects is used for generation or programming [13].

As already mentioned, a PoC for DIPs in practice has been developed and is presented as an example in Sections 3 and 4: Section 3 takes a closer look at DIP definition and the model-driven infrastructure (DIP Profile, DI app generator), and Section 4 presents the application of DIPs in the context of the methodology (CIM, PIM, PSI: transformation and generation).

However, before this, another important aspect needs to be introduced: since DIPs provide solutions for problems with mappings and transformations of data elements, they are inherently structural and data model oriented. However, data integration tasks usually include different software systems that act as a data source and a target so that process- and message-oriented (behavioral) aspects must also be considered. This is achieved with enterprise integration patterns (EIP [14]). The framework “Apache Camel” [15] is an implementation of EIPs, and is used for the DIP DSL (code generator and manual implementation). Section 3.4 describes EIPs after the topics “Data Model” (Section 3.1), “Data Integration” (Section 3.2), and “Data Integration Patterns” (Section 3.3) are covered.

## 2. Related Work

Data integration is described in the literature [16]. General topics are technologies, architecture, and components of data integration systems, schema and data matching, mappings, mapping languages, constraints and integrity rules for data integration, query processing for data integration, wrappers, and other aspects such as data warehousing, knowledge representation, and data provenance for data integration. Data integration patterns are mentioned in some books but only on an architectural and abstract level, for instance, [17], with the patterns “loose coupling”, “hub and spoke”, “synchronous and asynchronous interaction”, “request and reply”, “publish and subscribe”, and “two-phase commit”. Other high-level patterns can be found from software vendors for system integration platforms [18]. The patterns “Migration”, “Broadcast”, “Bi-directional sync.”, “Correlation”, and “Aggregation” are described. These high-level patterns can be classified as heuristics; they are defined using only natural language (no formal or semi-formal definitions or pattern description format).

Other authors focus on certain aspects such as Big Data and the underlying theory [19] covering topics such as schema mapping and transformation in conjunction with second-order tgds and operational semantics for database mappings. Additionally, domain-specific data integration literature exists, e.g., data integration for life sciences [20].

Neither the general literature about data integration (one book with the promising title “Data Integration Patterns—a Complete Guide” does not contain a single pattern. It is just a collection of questions for self-assessment [21]), nor specific books, journals, or proceedings contain data integration patterns on a formal or semi-formal level, so that they can be used in a model-driven approach.

Other pattern catalogs such as software design patterns, enterprise integration patterns, or data modeling patterns, do not cover the specifics of data integration, for instance, the mapping of data elements.

Commercial and non-commercial/open-source software libraries and integration platforms for data integration are available. Examples are the Java library “Modelmapper” for (semi-)automatic mapping of Java classes [22] or Jitterbit Harmony platform for enterprise integration [23]. Some software products offer a broad spectrum of features including data element mapping and data flows modeling, but detailed and formally defined DIPs (see next section) cannot be found.

The combination of patterns and model-driven software development such as MDA is described in various books and articles. Two examples are: [24], which presents a pattern-based model driven architecture approach, and [25], which introduces REA (resources, events, agents) patterns (business patterns) in connection with modeling techniques. Data integration is not covered on a detailed level (see Section 3.3).

### 3. Data Integration and Data Integration Patterns

Sections 3.1 and 3.2 explain fundamental aspects for data integration in the context of enterprise computing such as data models for enterprise applications, the problems with multiple operational data models and data stores, and definition of data integration. Sections 3.3 and 3.4 are dedicated to patterns for data and enterprise integration. Section 4 presents the model-driven and pattern-oriented approach in detail, with an example.

#### 3.1. Data Models for Enterprise Software Systems

Enterprise software systems (or enterprise applications) support large parts of a business in terms of data and processes. Their inherent complexity leads to non-trivial data models. Additionally, businesses often need more than one enterprise software product so that several data persistence layers (usually database systems) are in operation. To make things worse, these different systems are heterogeneous in terms of data when their underlying data models are not (conceptually or technologically) compatible. An integrated view on enterprise data is needed in the following cases: (1) daily operational tasks where more than one software system is involved; (2) tactical tasks that include when decisions and data from different systems are needed; and (3) strategic work that needs the “big picture” of a business. “An Enterprise Data Model is an integrated view of the data produced and consumed across an entire organization” [26]. That does not necessarily mean that enterprise data must be stored in one (physical or logical) system, e.g., one single relational database. There are many different ways to establish and implement an enterprise data model (EDM). Here are two examples:

- A data warehouse (DW) contains all the data from heterogeneous data sources (enterprise applications). A DW can store data in a dimensional or normalized manner, centralized or decentralized, and in different formats, but in either way, it provides an integrated view on data, often through OLAP, e.g., drill-down, dicing, slicing, roll-up, or pivoting. Data warehousing is a uni-directional approach: data is extracted from operational systems to the DW, but not vice versa;
- A system that follows a micro-service architectural approach and implements the “Database per service” patterns [27] to store data in a decentralized way. The integrated view can be archived with the “Command Query Responsibility Segregation (CQRS)” pattern on the database level and the “API Composition” pattern with in-memory data merging.

A data model is often confused with a domain model. A domain model describes an existing domain or business area, and belongs to the problem model. A data model is part of the (software) solution and “covers” the domain. It is important to notice that a domain often consists of subdomains, for instance, accounting, manufacturing, and sales.

Among the many aspects of data models and data modeling, the most important are logical and physical models, the data modeling paradigm. Examples are relational, multi-dimensional, or object-oriented, data modeling language (or notation), and OLAP or OLTP. Furthermore, differences and variations are significant even within these aspects, e.g., data normalization is an essential characteristic of a data model for an (operational) enterprise system that uses a transactional, relational database. However, in the context of a data warehouse, this is not the case. Data modeling patterns [28] provide valuable solutions for common modeling problems, but they are not identical to data integration patterns: Sections 3.2 and 3.3 will explain this further.

#### 3.2. Data Integration: Introduction and Definition

Data integration leads to a unified, consolidated, or integrated view on data from different data sources. This includes not only technological aspects such as querying data from different databases and merging the results, but also semantic and conceptual aspects because combining or integrating data often requires analyzing and understanding data models to be able to map or transform data. Experts who manage data and are in charge of data integration need a deep knowledge about the domain for which the data is being

used. Therefore, the specification and implementation of data integration systems require human expertise and involvement.

In the scientific literature, definitions of data integration often comprise a mediated or global schema (or enterprise data model on the conceptual level, see also [16,29]), that allows queries on the integrated data. Source and global schema need mappings or transformation rules so that source data can be represented by a global schema. Figure 2 shows the basic architecture of an integration layer with a consolidated and integrated enterprise database and a global schema (enterprise data model). Examples of data sources are databases of application software systems such as ERM or CRM, files on a local PC, or APIs (application programming interfaces) of cloud apps. Connectors or wrappers are used to transfer the data from the sources to the integration layer.

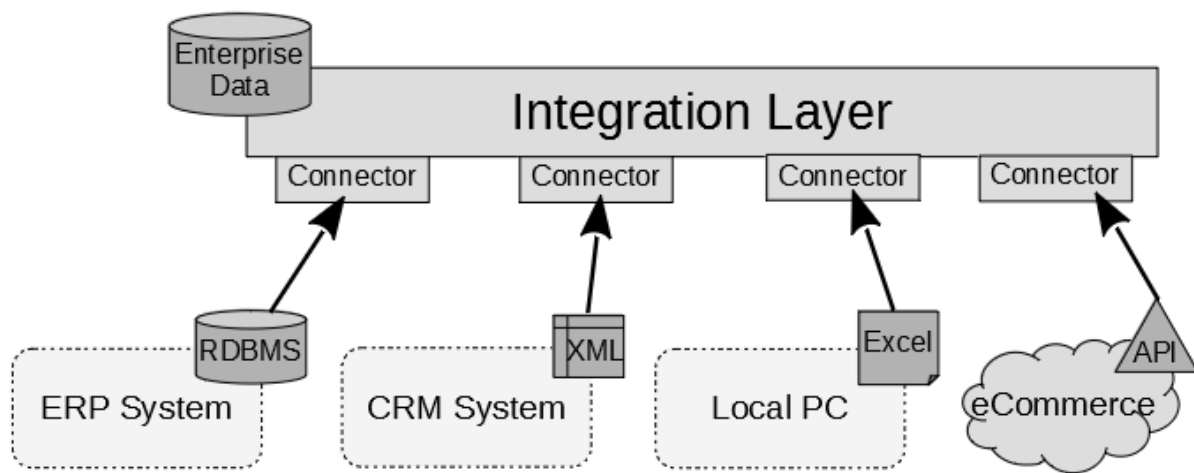


Figure 2. Data integration layer: basic architecture.

However, it must be pointed out that data integration can also be established without one single global schema or enterprise data model. The concept of bounded contexts in DDD (domain-driven design) is a good example: it has its own (ubiquitous) language and “delimits the applicability of a particular model” [30] (p. 336). Furthermore, a bounded context can have relationships to other bounded contexts, but is not identical to a subdomain or always has a 1:1 relationship [31]. Figure 3 exemplifies the concept of bounded contexts and shared concepts: the sales context has customers and articles, and the accounting context consists of customers and products.

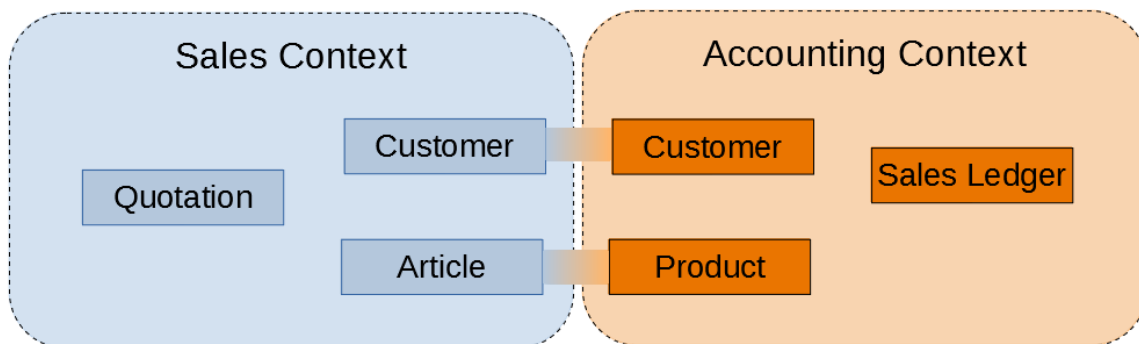


Figure 3. Two bounded contexts in DDD (sales and accounting) with shared concepts for customers and products (articles).

Integration is possible through relationships between bounded contexts. A context map and integration patterns (integration patterns are shared kernel, customer/supplier, conformist, and anti-corruption layer) are used for this purpose [32] (Chapter 4). Bounded

contexts can be used for microservice architectural design of software systems where a “microservice should not span more than one bounded context” [33].

In the end, a more general definition of data integration than the one given in Section 1 is necessary so that not only enterprise systems as a whole, but also microservices or other software components and even humans, are included:

**Definition 1.** *Data integration consolidates data from different sources and enables interactions between multiple software systems, services, or humans to interchange data or provide a unique view on heterogeneous data for a target. Software systems, services, and humans can act as sources or targets for data integration tasks. Sources and targets can be identical.*

### 3.3. Data Integration Patterns

One core concept of data integration patterns (DIP) is mapping source data elements such as relational tables and columns to target data elements such as classes and properties. There are two fundamentally different types of data integration levels where DIPs apply: (1) modeling, i.e., creating or modifying a data model, and (2) transferring data elements (stored in different data models). In this context, the latter is relevant—the data models exist and cannot be changed.

Source and target data models are usually different and incompatible in terms of mappings. An example is a mapping from a relational data model that uses foreign keys [34] to an object-oriented class model with object references [35,36]. Mappings of data conversion and transfer tasks can be simple, e.g., from a field “person\_name” to another field “persName”, but also quite complex, for instance, the splitting of data from a relational table and conversion to objects of classes of an inheritance tree. Additionally, constraints or consistency rules might exist and lead to errors, such as inconsistencies, during mapping and transformation operations.

Data mapping takes place on field or entity level, has a multiplicity (1:1, n:1, n:m), is uni- or bi-directional, and is transactional or non-transactional. Complex mappings of inheritance trees and associations require special rules, such as grouping of field values of a data structure and transferring them as data sets of a different data type that is associated with the original data structure. This is similar to the “Extract Class” refactoring pattern [37].

Table 1 shows some DIPs that are used in data integration specifications for relational or object-oriented data models. Each pattern is specified in detail with information such as name, type, objective, and problem (the complete descriptions are beyond the scope of this paper). The term “field” is used to refer either to a table column (relational data) or a property (object-oriented class).

In order to describe a DIP in detail, a well-known format is used that includes pattern name, classification, problem/motivation, solution, consequences, sample code, and related patterns [38] (pp. 3, 6) adapted to the data integration domain. The classification does not follow a strict flat or hierarchical structure (taxonomy), i.e., one pattern can belong to multiple classifications or more than one category.

The following table shows an example of a detailed description of the DIP “Split Field”: the input is a field value from an object-oriented class or a relational table (source), and the output is two or more values for field of another class or table (target).

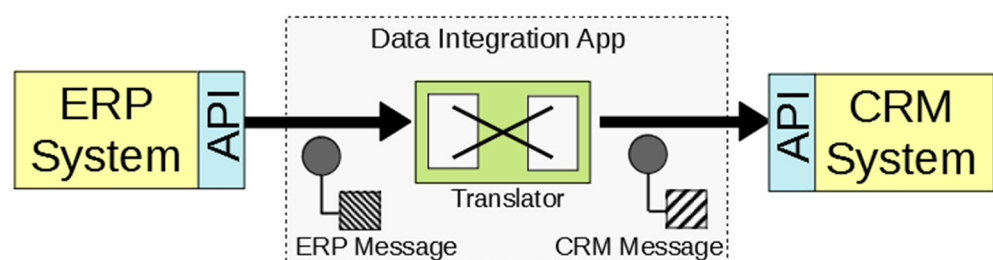
### 3.4. Enterprise Integration Patterns

While DIPs focus on the data aspect of integration, enterprise integration patterns (EIP, [14]) target the messaging between enterprise systems: “Messaging is a technology that enables high-speed, asynchronous, program-to-program communication with reliable delivery” [14] (p. xxxi). In other words, EIPs cover the process-oriented aspects of communication between systems and services. Therefore, these patterns answer the question “how is communication handled?”, and DIPs describe the data-oriented aspects when communication via messaging takes place, answering the question “how is data handled?”.

**Table 1.** Data integration patterns: source on the left side (orange) and target on the right side (green) [5].

Pattern Name	Icon	Description
Field-to-Field		A field of a relation (or class) maps to a field of another relation or class.
Merge Fields		Multiple field values are merged into one single field value.
Split Field		A single field value that contains multiple data elements is split into values for other different fields.
Aggregate Field		A field value is aggregated and stored in another field.
Convert-Field-to-Enumeration		A field value is used to determine an enumeration literal.
Relations-to-Inheritance-Tree		Data from a group of relations is mapped to objects of an OO inheritance tree.
Object-to-Object		A data object (data set, row, class instance) is mapped to another object (implies field mappings, see Field-to-Field pattern). The plus symbolizes that new objects are created if they do not exist.
Relations-to-Association-Class		Data from relations are mapped into objects (with an n:m association and an association class).
Extract-Fields-to-Class		A group of fields in one or more relations maps to an object of a class.
Field-to-Relationship		A field value is used to create a relationship (or association) between objects.
Target-object-to-relationship		An object of the target data model is used as a reference. A new relationship must be created in an object of a class (or relation).

The translator is an example of an EIP. In the context of a message exchange between two systems or services, the message needs to be translated (due to different data formats, encryption, API, and DB schema). Figure 4 shows this scenario with an example where an ERP system provides API endpoints serving as a source and a CRM system with different API endpoints serving as a target. A data integration application connects these two systems, taking an ERP message and translating it into a CRM message, i.e., the EIP translator is being applied in this example. For the EIP, the message and the actual translation is a black-box, i.e., there is no further description of message or data translation mechanism. This is the reason why DIPs can be helpful to specify the details of this data translation, e.g., with a Field-to-Field pattern or a Split Field pattern.



**Figure 4.** EIP Translator (on the basis of [14] (p. 86)).



Enterprise integration can be specified with a pipeline where more than one EIP is applied. Figure 5 exemplifies a pipeline: an input message from the ERP system contains a list of addresses that is processed by a Splitter EIP [14] (p. 259); the splitter splits the message and provides addresses of the list as single message so that a translator can process the addresses individually; an aggregator collects all outgoing addresses from the translator and creates one single message containing a list of CRM addresses; this aggregated message is then transferred to the CRM system. The translator is being used in a loop-like process because of the Aggregator EIP [14] (p. 268). Three EIP were applied in this integration pipeline.

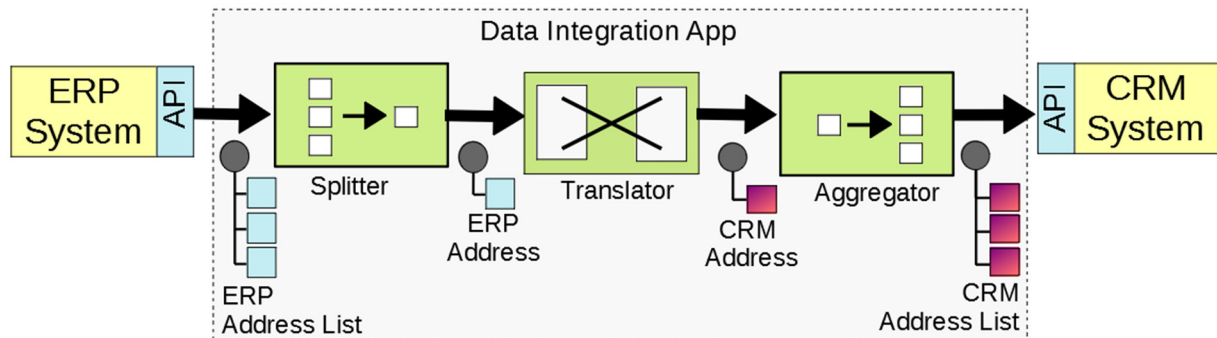


Figure 5. Integration pipeline with three EIPs (Splitter, Translator, and Aggregator).

In Section 3.3 (s. Table 2, example code) Apache Camel was used in order to provide a communication route between the ERP and the CRM system so that the data integration of products could be implemented. As already mentioned in Section 1, Apache Camel implements EIPs, and therefore, it is an ideal platform for the implementation of DIPs. Section 4 will explain in detail the “symbiotic” combination of DIPs with EIPs.

Table 2. Detailed pattern definition for data integration pattern “Split Field”.


Pattern Name	Split Field	
Description	A single field value that contains merged data or multiple data elements is split into values for separate fields.	
Also Known As	Compound Field Split, Normalize Field Data, Demerge Field Value	
Classifications	<ul style="list-style-type: none"> <li>Modeling paradigm: object-oriented, relational;</li> <li>Context: field(s) of table or class.</li> </ul>	
Problem/Motivation	A field from a source data element contains more than one chunk of information. Therefore, these chunks or parts must be transferred into different fields of a target data element. This happens, for instance, with denormalized data or semi-structured data.	
Solution	<p>The field value of the source data elements must be split so that the parts can be extracted and transferred separately as field values of the target data element fields. Splitting is performed depending on the data type, format, and structure:</p> <ul style="list-style-type: none"> <li>Data type: numerical (range, min, max), string (delimiter, index, regex), binary (bit mask, range), date/time (date separation from time, time zone, date part extract), image (color channel, area);</li> <li>Format: integer and decimal part of floating-point number, lines of a string, numerical and alphabetical characters of a string;</li> <li>Structure, e.g., JSON, XML, HTML, and XHTML.</li> </ul>	
Consequences	The rules for field splitting can be complex and error-prone. Especially in the case of poor data quality, splitting can run into error situations that need to be addressed. Constraints and data quality rules for the target field might need to be checked first.	

Table 2. Cont.

Pattern Name	Split Field	
Sample Model	<p>The example below (domain: clothing) shows the source element (class “ErpProduct”, property “category”) on the left side and the target elements (class “CrmProduct”, properties “quality”, “family”, and “size”) on the right side. Packages, classes, and properties are marked as source and target, respectively, but property marking would have been sufficient in this case. The DIP application is shown in the middle: a split field pattern is applied to the source field “category” (product); the source field value is split into three target field values: “quality”, “family”, and “size”.</p>	
Sample Code (Java)	<p>On the left side, the two classes “ErpProduct” and “CrmProduct” are defined (Lombok is used for constructor, getter, and setter code generation). On the right side, the DI within the Camel route is implemented: the field splitting is defined in the upper part of the configure method (object “diProductCategorySplit”) with the Java DSL for DIP and then embedded into an Apache Camel [15] route (route name “erpProduct2crmProduct”).</p> <pre> @NoArgsConstructor @Getter @Setter public class ErpProduct {     protected Integer id;     protected String name;     protected String category; } ...  @NoArgsConstructor @Getter @Setter public class CrmProduct {     protected Integer id;     protected String name;     protected ProductQuality quality;     protected String family;     protected Size size; } ...  public void configure() {     DI diProductCategorySplit = diBuilder         .layer(Layer.ApplicationSystem)         .service(this)         .mapping(SfMappingBuilder.class)         .algorithm(SplitAlgorithm.Delimiter, "_")         .source(ErpProduct.class.getMethod("getCategory"))         .target(CrmProduct.class.getMethod("setQuality"))         .target(CrmProduct.class.getMethod("setFamily"))         .target(CrmProduct.class.getMethod("setSize"))         .end();     build();      from("direct:erpProduct")         .routeId("erpProduct2crmProduct")         .setExchangePattern(ExchangePattern.InOnly)         .process(diProductCategorySplit)         .to("direct:crmProduct"); } ...                 </pre>	
Related patterns	If the target fields are not in one single data set or object, selectors and aggregators are needed.	

## 4. Model-Driven and Pattern-Oriented Method for Data Integration Tasks

### 4.1. Methodology Task Definition

As already stated, data integration almost always takes place in the context of service or enterprise application integration. System integration, in general, can be a complex task, and in the case of data integration, two different pattern types (EIPs and DIPs) must be applied. That is why a method was introduced in Section 1.1 (Methodology Overview), to give the system/data engineer a proper guideline for real-life projects. It is important to note that the focus here is more on the technological parts, and it is assumed that business needs and processes, data modeling, and requirements analysis are already analyzed, specified, and documented. The following steps are a high-level description of the method, with the first two steps leading to the CIM, steps three and four leading to the PIM, and steps five and six leading to an implemented (and executable) integration application (standalone or as part of a bigger system).

1. Verification of prerequisites: ensure that the necessary information about the systems and services, including the data models and business processes, is available, up-to-date, and correct;
2. Identification of data integration tasks: based on the (business) processes and the source and target systems and services, tasks for data integration are identified. This activity leads to a list of potential data integration relevant tasks that also include the process-oriented context;
3. Analyze, break down, and specify data integration tasks in connection with communication and messaging: some DI tasks are simple, but others are complex and consist of communication routing or logic, conversion, translation, or similar. The tasks should be analyzed and described. Complex tasks should be broken down into smaller tasks. Not only is the data integration important in the narrow sense, but also its context is important during this step, e.g., the communication mechanisms between systems or services;
4. Application of EIPs and DIPs: this is probably one of the most important steps but can also be challenging because it requires knowledge about the domain, the processes, the data, and the patterns. After the messaging is clarified (application of EIPs), the data-oriented aspects can be addressed, and DIPs are applied. Sometimes the data aspect is so dominant that EIPs and DIPs must be used as a “package”, i.e., they build one inseparable compound pattern;
5. Implement or generate integration: during implementation (or generation of the code), the system and data integration should be simulated or tested based on the specified tests. Code coverage is insufficient because of the relevance of the data. Therefore, data element coverage is as relevant as code coverage;
6. Test scenarios and test cases: tests should be created as early as possible. Negative and positive tests, equivalence partitioning, boundary testing, and decision table testing are some examples of relevant test techniques for data integration projects.

Other aspects and cross-cutting concerns are omitted in the steps described above, but they play an important role. System and service integration is often not isolated, but part of a larger project. Topics such as code quality, security, error handling, logging, monitoring, and performance engineering, must be considered as soon as possible.

The generic method steps described above are just an overview, and do not represent a ready-to-be-used method. Nevertheless, this bird’s-eye perspective is necessary because it points the spotlight on the activities needed for data integration. As the word “generic” implies, specific notations or languages are not prescribed (except the EIP and DIP application) so that the steps can easily be integrated into various project management techniques such as Scrum. However, for data modeling, UML or ERM are recommended.

A concrete method for a data integration project can only be defined when the project context is clear. For instance, a software project that develops a software system based on microservice architecture has a different viewpoint on data integration than a project that develops an integration hub or a middleware for the integration of all the monolithic legacy enterprise systems of a business, or a project for data warehousing that deals with ETL (extract, transform, load) and OLAP [39,40].

It is clear that not all possible scenarios can be described here. Instead, one fictional but realistic case will be used as a concrete example to avoid going beyond the scope of this paper, namely, two legacy systems with a relational data model must be integrated. The focus is on the conversion of customer data and their addresses. A concrete methodology for this data integration project will be shown in detail during the presentation of the example in the following sections.

Section 4.2 starts with the formal definition of DIPs on the meta-level (M2) and the implementation of a code generator. Section 4.3 then covers the data model and integration analysis (CIM) and data integration specification (PIM), and Section 4.4 takes a look at the data integration implementation.

### 4.2. DIP Definitions

For the formal definition of DIPs, the UML profiling mechanism was used. Figure 6 shows a part of the DIP UML profile that defines a set of stereotypes extending the UML meta-classes “Package”, “Class”, “Dependency”, and “Property” (from the UML class model). The stereotype “Datastore” is the base class for “Source” and “Target” needed for “DataIntegration” that represents a formal data integration specification with a UML package. A datastore can be either a whole package or a single class. The constraint for “DataIntegration” was defined in OCL (object constraint language, [41]) and enforces that at least a source or a target for the data integration specification exists.

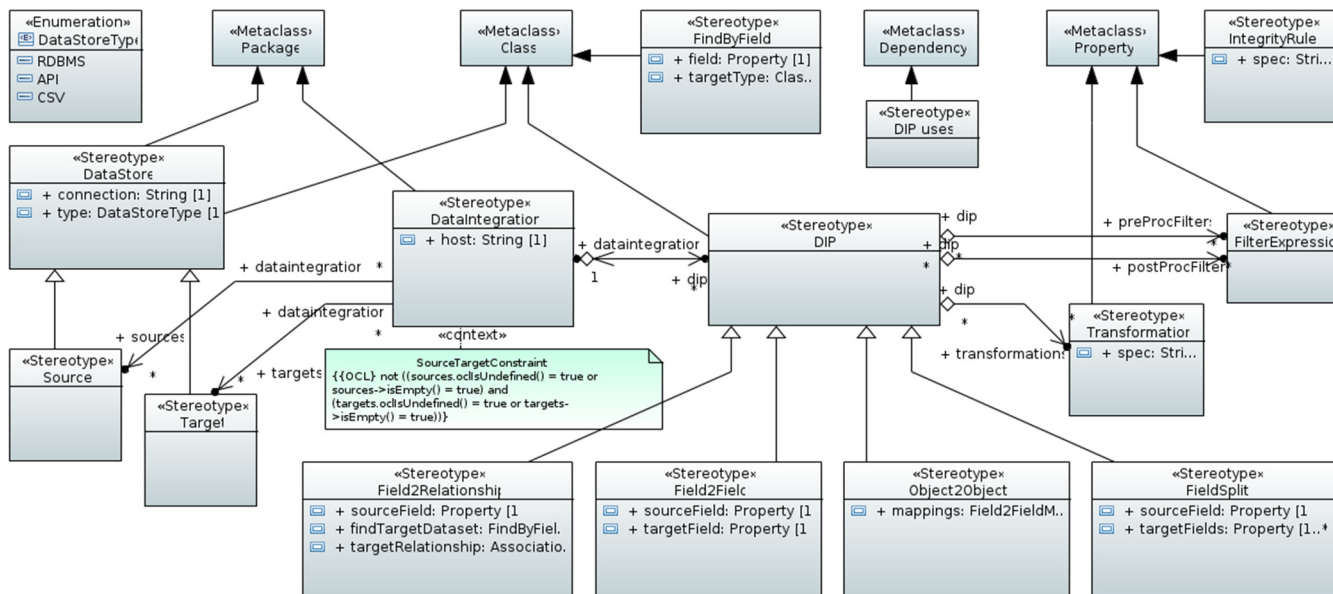


Figure 6. DIP Profile (definition of stereotypes and tagged values) on the meta-level (M2).

The stereotype “DIP” is a base class for concrete DIPs such as “Field2Relationship”, “Field2Field”, “Object2Object”, and “FieldSplit”. Every DIP belongs to a data integration specification and has filters and transformations. DIPs can have properties (tagged values) that are needed when applied. In the case of the DIP “Field2Relationship”, a finder mechanism must be specified to identify the referenced object. The stereotype “FindByField” was used in this context. DIPs can have dependencies on other model elements. For this purpose, the “DIP uses” stereotype that extends the meta-class “Dependency” was part of the profile.

The profile for EIPs was simple and defined only the patterns that were necessary for the example. Figure 7 shows the EIPs “Aggregator”, “Splitter”, and “Translator”. The stereotypes “In” and “Out” were used for the marking of data sources and targets.

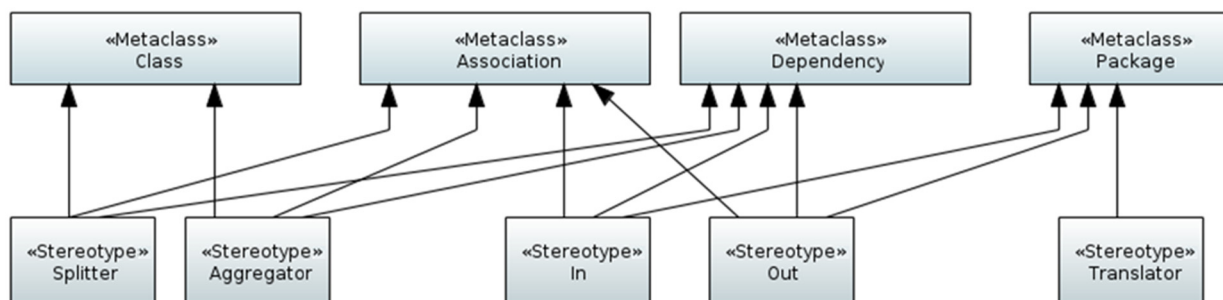


Figure 7. EIP Profile.

Profiles can be used for model transformations (PIM-to-PSM) and code or text generators (PIM/PSM-to-PSI). In this showcase, a direct code generation strategy was chosen to keep the model-driven approach simple. Applied stereotypes within a model on the model-level (M1) were used in the code generation templates of the code generator. The Eclipse project Obeo Aceleo that implements OMG's MOF2T specification was used [10,11].

Figure 8 shows a part of the code generator for data integration applications. Two code generation templates are shown: on the left side is the main template that iterates over all packages of the data integration specification, it calls other templates for data store classes—EIPs or DIPs; on the upper right side is the data store template for the generation of Java classes that stores data such as customers or addresses; on the lower right side, a generated Java class (data store for ERP addresses) is shown.

```

generate.mtl
2 1**
3  * Module generate: Main code generation templates for package and class iteration
4  */
5  [module generate('http://www.eclipse.org/ocl/1.1.0/UML', 'http://www.eclipse.org/uml2/5.0.0/Types', 'http://
6  [import DIP_Generator:util::ProfileUtil /]
7  [import DIP_Generator:util::EipUtil /]
8  [import DIP_Generator:util::DipUtil /]
9  [import DIP_Generator:util::DataStoreUtil /]
10
11 1**
12  * Entry point for the code generator for data integration applications.
13  * All packages of the model are analyzed
14  * @param model root element of the DI specification contains schemas (datastore) and DI specifications
15  */
16  [template public generateDataIntegration(model : Model)]
17  [comment @main/]
18  [file ('Model' + model.name + '.txt', false, 'UTF-8')] [comment log the processed package in txt file/]
19  [comment iterate over all packages in the model/]
20  [for (p : Package | model.ownedElement->select(oclIsTypeOf(Package)))]
21  [generateDataIntegration(p)]
22  [/for]
23  [/file]
24  [/template]
25
26 1**
27  * Template for package processing.
28  * @param model root element of the DI specification contains schemas (datastore) and DI specifications
29  */
30  [template public generateDataIntegration(pkg : Package)]
31  [comment debug/] [getStereotypes(pkg)/]
32  [if (hasAppliedStereotype(pkg, 'DataIntegration') = true)]
33  [let className : String = 'Route'.concat(pkg.name)]
34  DI Package: [pkg.name/] ([pkg.namespace.name/])
35  [file (className.concat('.java'), false, 'UTF-8')]
36  [generateRouteHeader(className)]
37  [for (pkgEIP: Package | pkg.ownedElement->select(p | p.oclIsTypeOf(Package)))]
38  [getStereotypes(pkgEIP)/]
39  [/for]
40  [for (pkgEIP: Package | pkg.ownedElement->select(p | p.oclIsTypeOf(Package) and hasAppliedStereotype(p, 'DataStore')))]
41  [generateRoute(pkgEIP)/]
42  [/for]
43  [/file]
44  [/let]
45  [elseif (hasAppliedStereotype(pkg, 'DataStore') = true)]
46
DataStoreUtil.mtl
2  [module DataStoreUtil('http://www.eclipse.org/uml2/5.0.0/UML')]
3
4 1**
5  * Iterates through all classes in a package and calls the
6  * generation template for data store classes (source or target)
7  * @param pkg package with data store classes
8  */
9  [template public generateDataStore(pkg : Package)]
10 [for (class : Class | pkg.ownedElement->select(oclIsTypeOf(Class)))]
11 [file (className.concat('.java'), false, 'UTF-8')]
12 [generateDataStore(class)/]
13 [/file]
14 [/for]
15 [for (innerPkg: Package | pkg.ownedElement->select(p | p.oclIsTypeOf(Package)))]
16 [generateDataStore(innerPkg)/]
17 [/for]
18 [/template]
19
20 1**
21  * Generation of data store classes (source or target)
22  * @param class data store class
23  */
24  [template public generateDataStore(class : Class)]
25  [let className : String = class.name]
26  import lombok.NoArgsConstructor;
27  import lombok.Getter;
28  import lombok.Setter;
29
ErpAddress.java
1  import lombok.NoArgsConstructor;
2  import lombok.Getter;
3  import lombok.Setter;
4
5  @NoArgsConstructor @Getter @Setter
6  public class ErpAddress {
7  protected String addressId;
8  protected String streetAddress1;
9  protected String streetAddress2;
10 protected String city;
11 protected String country;
12 protected String zip;
13 }
14

```

Figure 8. Aceleo code generation templates for data integration application.

The code generation templates demonstrate the principle of using meta-classes of UML class model and stereotypes defined in profiles. Therefore, the profiling mechanism extends the meta-level (M2) of the modeling language, and the definition of code generation templates can be considered meta-programming.

### 4.3. Analysis and Specification for Data Integration

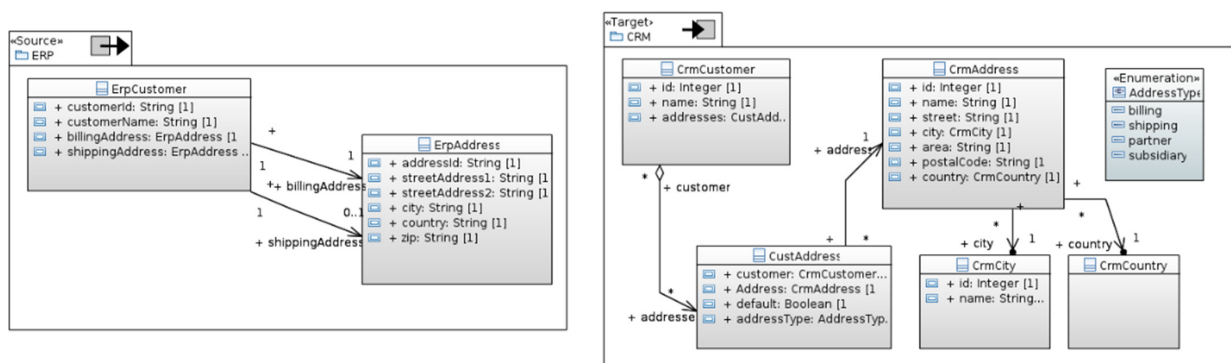
#### 4.3.1. Data Model and Integration Analysis

This was the first part of a data integration project. The focus was on the analysis using the example of customer and address integration. The steps were as follows:

1. Business needs and processes, data modeling, and requirements analysis: Customer data created in the ERP system must be transferred to the CRM system in real-time. We assumed that the two systems and the APIs were known, data structures and constraints were documented, and process-oriented aspects and cross-cutting concerns, e.g., the triggers for the data transfer, security mechanism, and data quality, were also specified;
2. Cross-cutting concerns: it was assumed that these topics were covered in step 1;
3. Verification of prerequisites: as described in Section 4.1 (Overview), API documentation, process description, and a (relational) data model existed. Figure 8 shows the data models of the source system (ERP) and the target system (CRM);

4. Identification data integration tasks: the data integration in this example was uni-directional, transactional, and in real-time (in practice, a question should be raised as to whether the data integration would be better as bi-directional, because both systems (ERP and CRM) are operational application systems where data can be created and modified. However, this discussion is outside the scope of this paper). The main data integration tasks were:
  - (a) Transfer of customer master data from ERP to CRM;
  - (b) Transfer of customer addresses ERP to CRM;
  - (c) Creation of classification of customer addresses: billing (invoice) or shipping (delivery).
5. Test scenarios and test cases (1): the processes, data structures, and data integration tasks were clarified on a level that allowed us to specify the test scenarios and commence the creation of the test case;
6. Analysis, break down, and specification of data integration tasks in connection with communication and messaging: The data integration tasks from step 3 were of simple or medium complexity.

The data models need to be briefly discussed for the exemplification of the subsequent application of DIPs in connection with EIPs in the next section. The simplified legacy relational data model from an ERP system served as the source model: a customer had a billing address (mandatory) and an optional shipping address (see Figure 9).



**Figure 9.** ERP source data model (left) and CRM target data model (right) with associations also shown as properties in the class definition.

The CRM system was based on a more powerful data model: a customer could have more than one billing and shipping address (including a default address), and an address relationship had more types (billing, shipping, partner, subsidiary). Other structural differences were the city and country tables in the CRM model, that did not exist in the ERP system. Both data models were also incompatible on the level of data types for identical properties. An example was id fields, which were strings in the ERP data model and integer values in the CRM data model.

The data models described above were part of step one and three of the method. The data models were modeled using UML class diagrams. There was no inheritance involved so that the models could be interpreted as relational data models. Therefore, for the sake of simplicity, we used the object-oriented terms “class” (for relational table) and “object” (for a table row or data set).

Step 4 leads to the identification data integration tasks. Table 3 describes these steps in detail. Neither EIPs nor DIPs needed to be applied at this point, that focused on understanding the integration tasks (as part of the problem space) and not the solution. The description could be non-formal or semi-formal, and the integration tasks could be of different complexity levels because the main goal of this step was the development of a clear understanding of the data sources and targets and the activities necessary for data integration.

**Table 3.** Step 3: Identification of data integration tasks for the example.

Pos.	Name	Source(s)	Target(s)	Activity	Description
(a)	Customer master data	ERP System: ErpCustomer	CRM System: CrmCustomer	Transfer all objects (one by one)	For every object of ErpCustomer, a new object of CrmCustomer is created. The properties id and name are transferred. For the address transfer, see task b.
(b)	Customer addresses	ERP System: ErpCustomer, ErpAddress	CRM System: CrmCustomer, CustAddress, CrmAddress, CrmCity, CrmCountry	1. Transfer all objects from ErpAddress to CrmAddress (one by one) 2. For each billing and shipping address create CustAddress	1. For every object of ErpAddress, a new object of CrmAddress is created. All properties are transferred. 2. For each value of billingAddress and shippingAddress a new object of CustAddress is created (referenced to the address). The correct addressType must be set (see task c).
(c)	Address type	ERP System: ErpCustomer	CRM System: CustAddress	Set the field value for addressType	During the creation of CustAddress objects (see b, 2.), the address type must be determined (depending on the source data) and set in the CustAddress object.

During step six, data integration tasks must be analyzed, broken down (if too complex), and specified in connection with communication and messaging. Task a was a simple mapping between properties (table columns). The only conversion was the id from string to integer. A conversion logic or an error handling must be defined for the case when a string could not be converted into an integer because it contained non-numeric data.

Task b was too complex and needed to be split into smaller integration tasks, starting with the first part (transfer all objects of “ErpAddress” to “CrmAddress” objects), which required:

- The performance of a simple mapping for the following properties: “id” (see task a), “streetAddress1”, and “zip”. The property “streetAddress2” could be mapped to “area” (but this might need further analysis);
- For the properties “city” and “country”, the corresponding objects in the target system needed to be found prior to them being assigned to CrmAddress object properties. If a city or country was not found, it must be created.

The following data integration tasks (second part of task b and task c) shared the same process-oriented context (iterate over all ErpAddress objects) and must be processed in a specific order:

1. For a billing address in an object of ErpCustomer, a CustAddress object must be created and the address type set to “billing”. Proceed analogously for the shipping address;
2. The object for CrmAddress determined based on the billing or shipping address in the ErpCustomer object and the address set in the CustAddress object;
3. The newly created CustAddress object added to the list of addresses of the CrmCustomer object.

To summarize the results of this step, Table 4 shows a modified list of data integration tasks. A new column is introduced for the process-oriented aspect because some tasks need a common process logic, e.g., iterate over all objects. The list shows six different data integration tasks that must be performed in three different process contexts.

The data integration tasks shown in Table 4 could be further improved: process contexts 1 and 3 are almost identical because an iteration over all ErpCustomer objects is necessary in both cases. Therefore, the tasks should be rearranged: (b) and (c) should be processed first (in process context 1) and then (a), (d), (e), and (f) within another process context. The next section takes this into consideration.

**Table 4.** Step 6: Analyze, break down, and specify data integration tasks for the example.

Process	Pos.	Name	Source(s)	Target(s)	Activity	Description
1. Erp Customer Iteration	(a)	Customer master data	ERP System: ErpCustomer	CRM System: CrmCustomer	Transfer all objects	For every object of ErpCustomer, a new object of CrmCustomer is created. Properties id and name are transferred.
	(b)	Customer addresses	ERP System: ErpAddress	CRM System: CrmAddress	Transfer all objects	For every object of ErpAddress, a new object of CrmAddress is created. Properties id, streetAddress1, streetAddress2, and zip code are transferred.
2. Erp Address Iteration	(c)	City and Country for customer address	ERP System: ErpAddress	CRM System: CrmAddress, CrmCity, CrmCountry	Set properties	Find city and country in the target system and assigned to CrmAddress object. If a city or country is not found it must be created.
	(d)	Association object for customer and address	ERP System: ErpCustomer	CRM System: CustAddress	Transfer all objects	Create CustAddress object for each billing and shipping address and set the address type
3. Erp Customer Iteration: billing/shipping address	(e)	Set the address in the association object	ERP System: ErpCustomer	CRM System: CrmAddress, CustAddress	Set properties	Determine CrmAddress object. Set the address in the CustAddress object.
	(f)	Add association object to customer address list	ERP System: ErpCustomer	CRM System: CrmCustomer, CustAddress	Add object to list	Add CustAddress object to list of addresses.

#### 4.3.2. Data Integration Specification

These steps of the data integration example projects lead to formal data integration specifications for customer and address data from ERP to CRM through the application of EIPs and DIPs. In terms of the MDA the outcome is a PIM. The following steps are necessary (numbering continues from previous steps, see Section 4.3.1. Data Model and Integration Analysis):

7. Application of EIPs and DIPs: for the analyzed and specified data integration tasks, various EIPs and DIPs can be used;
8. Test scenarios and test cases (2): After finishing step 7, the details should be sufficient so that the test case creation can be finalized and test data can be generated or manually created. In addition, the testbed should be implemented (not described here).

Step 7 applies EIPs and DIPs for the analyzed and specified data integration tasks. Table 5 shows data integration tasks with applied patterns: lists of addresses and customers of the data source are processed so that the object of the target data model is created.

Figure 10 shows a part of the example as an UML class diagram (ERP addresses are transferred to CRM addresses). The data integration specification was a package containing one or more other packages that represent integration processes. In this example, a translator with the name “ErpAddress2CrmAddress” contained two data integrations modeled as classes marked with DIPs (“Address2Address” and “City2City”). The EIPs “Splitter” and “Aggregator” were applied for the data source and data target. The applied DIPs were details of a translator: Object-to-Object and Field-to-Relationship.

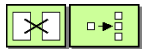
















The formal specification for the data integration could be used to generate or implement a data integration software application (or component) that performs the integration on the instance or runtime level (M0).

#### 4.4. Data Integration Implementation

A data integration software system can be implemented manually or generated. In this case, the code generator introduced in Section 4.2 will be used. The target architecture for the PSI consists of Java SE, Apache Camel, and the DIP DSL. The source artifact is the UML class diagram (with applied EIPs and DIPs).



**Table 5.** Step 7: Application of EIPs and DIPs for the data integration tasks (Icons for EIPs taken from [42]).

Process	Pos.	Name	EIP	DIP	Description
1. ERP Address Iteration	(a)	Customer addresses		Object-to-Object  Field-to-Field 	A splitter EIP allows the data integration logic to process each customer address separately. Object-to-Object and Field-to-Field DIPs are applied (in the context of the Translator EIP) for the address object (the plus on the target side means that new objects are created).
	(b)	City and Country for customer address	Translator 	Aggregator  Field-to-Relationship 	A Field-to-Relationship DIP processes a field value, e.g., a foreign key, and finds an object of a relation or class in the target data model. It then creates a reference to that object in another target object. An aggregator EIP combines the addresses to a list that is delivered to the recipient.
2. ERP Customer Iteration	(c)	Customer master data	Splitter Translator 	Object-to-Object  Field-to-Field 	A customer list is split into single objects with the EIP splitter. Customer objects are created with Object-to-Object and Field-to-Field DIPs.
	(d)	Association object for customer and address	Splitter Translator 	Object-to-Object  Field-to-Field 	A customer has to be split (EIP splitter): billing and shipping address are processed (Translator EIP). For each address a new object (Object-to-Object DIP) is created with field mappings (Field-to-Field DIP).
	(e)	Set the address in the association object	Translator 	Field-to-Relationship 	A customer has to be split (EIP splitter): billing and shipping address are processed (Translator EIP). For each address a new object (Object-to-Object DIP) is created with field mappings (Field-to-Field DIP).
	(e)	Add association object to customer address list	Translator 	Aggregator  Target-Object-to-Relationship 	The address–customer object must be added to the customer’s addresses list. A Target-Object-to-Relationship DIP is used for this purpose. An aggregator EIP collects all customers and combines them into one single message for the recipient.

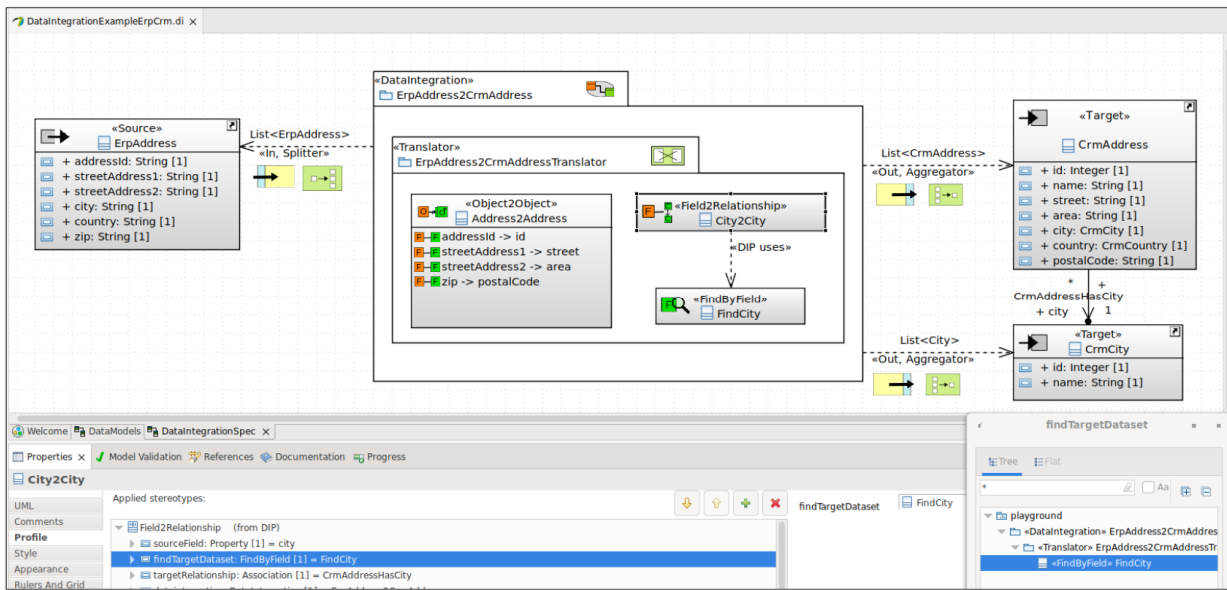


Figure 10. Visual model-based specification of the data integration example for customer addresses.

When continuing the numbered steps from the previous section, the next step is number nine: implement, generate, simulate, test. As already mentioned, the code generation is demonstrated here (simulation and testing is not described). Table 6 presents parts of the generated code for the first process context: the transfer of customer addresses from the ERP to the CRM system. As seen in the example for the DIP “Split Field” (s. Section 3.2, Table 2), the code consists of two parts: (1) the upper part is the definition of the data integration with applied DIPs, and (2) the lower part shows the Camel route definition (EIP).

Table 6. Step 9: Application of EIPs and DIPs for the data integration tasks.

EIP	DIP	Generated Code (Implementation)
		<pre>public void configure() throws Exception { //DI object creation with applied DIPs for data integration DI diErpAddress2CrmAddress = diBuilder .layer(Layer.ApplicationSystem) .service(this) .mappingObject2Object(true)//a) DIP Object-to-Object for customer addresses .source(ErpAddress.class) .target(CrmAddress.class) }</pre>
		<pre>.mappingsFF()//DIPs Field-to-Field .map("addressId", "id") .map("streetAddress1", "street") .map("streetAddress2", "area") .map("zip", "postalCode") .endMapping()</pre>
		<pre>.mappingField2Relationship()//b) DIP Field-to-Relationship for cities .map("city", "city") .find(City.class, "cities", "name") .endMapping() .mappingField2Relationship()//b) DIP Field-to-Relationship for countries .map("country", "country") .find(Country.class, "countries", "name") .endMapping() .endMapping() .build();</pre>
		<pre>from("direct:erpAddresses")//Apache Camel route (EIP) .routeId("erpAddresses") .setExchangePattern(ExchangePattern.InOnly) .split(body(), new AddressAggregationStrategy())//EIP splitter and aggregator .process(diErpAddress2CrmAddress)//EIP translator with DI object .to("direct:crmAddresses"); }</pre>

The implementation followed the data integration task definitions from step 7 in the previous section: All relevant parts can be identified in the code (see also the comments). The complete example with a test is available on Github [13].

## 5. Conclusions

The application of data integration patterns (DIP) in conjunction with enterprise integration pattern (EIP) shows that it is possible to provide a framework for complex data integration services that reduces the manual programming effort (through the Java DSL) or that even eliminates coding (through a visual model-driven approach). This proof of concept implementation of pattern and model-based data integration is promising.

The advantages of such an approach are: (1) a common language for system and data integration is introduced, that leads to better communication between the stakeholders of an integration project, (2) the data integration tasks are defined formally in a precise and concise manner with a visual notation (UML) eliminating the need for further documentations, and no programming knowledge is necessary because the data engineer is able to define and execute powerful data integration tasks without manual coding through code generation. Another advantage of the formal modeling approach is an easier quality assurance because simulations and tests can also be automated. Even if a manual implementation is needed, the coding based on this formal specification contributes to avoiding errors. The most interesting aspect is certainly automation—with this graphical or visual specification, the following options are available:

- Code generation: With OMG's MDA or other model-based software development techniques, the data integration specification can be considered a PIM (platform independent model) that can be transformed into a PSM (platform specific model) and to executable code (PSI, platform specific implementation). The generated code is a ready-to-be-used middleware for data integration based on Apache Camel and the DIP engine provided by the authors;
- Simulation: a simulator can take the data integration specification, generate test data (or real data) as an input, and create the output giving valuable information about the validity of the specification;
- Testing: test data and a test environment can be generated based on the specification, saving time and effort for the quality assurance tasks.

However, a debate about M2 is necessary. Questions such as “is a structural (or static) UML diagram sufficient for the definition of DIPs or are behavioral or dynamic diagram types or metamodels necessary?” should be answered.

The PIM-to-PSM transformation is omitted here in order to simplify the methodology and keep the example as compact as possible. A code generator was developed that creates code (PSI = platform specific implementation) directly from PIMs. This direct PIM-PSI approach lacks the flexibility that a PIM-to-PSM transformation has, but seems to be sufficient to evaluate the approach. Nevertheless, the creation of a PSM should be considered. For this purpose, QVT-Operational or QVT QVT-Relations are available for model-to-model transformations. Another open topic are executable models as an alternative to code generation. A data integration engine with an interpreter similar to xUML could be used in order to execute the UML-based data integration specifications.

Human factors also need more consideration: The experts who are in charge of process and data integration must be familiar with data modeling and EIP/DIP patterns. Open questions or topics for further research and development in this context are:

- The pattern catalog for data integration patterns must be extended. Additionally, the DIP engine that applies mapping specifications to real-world data integration scenarios needs more features. The visual editor for the graphical modeling requires a redesign. A community is needed in order to develop the necessary artifacts;
- Standards such as OMG's model driven message interoperability (MDMI, [43]) should be considered and used for the metamodel of the data integration approach presented

here. Constraint and consistency rules should be defined or visualized. OCL could be helpful for this task. The question here, is whether the experts accept these standards;

- More research for the methodology is necessary in order to provide guidelines in the context of real-life projects, e.g., the setup of an EDM (enterprise data management), tasks in the context of data migration, and development of microservices;
- Data quality and data cleansing are crucial aspects for data integration projects. These topics should be addressed in future work. A discussion with experts from different domains is crucial;
- Additionally, further research is necessary for the messaging and process-oriented aspects: the logic for the message passing had to be programmed in Java. Modeling the process (message passing), such as the chain of translators, splitters, and aggregators and using a process or workflow engine could add the necessary flexibility.

Code generation based on formal specifications is one of the most important aspects: the generated microservices for the data integration middleware can be instantiated and restarted after model changes have been made, leading to fast development cycles. Transformation and code generation frameworks such as OMG QVT [44] and MOFM2T [10] are used.

Other aspects such as Semantic Web technologies need further studies: “Semantic Web technologies have been used successfully in a wide array of domains such as health care and life sciences as a platform for information integration and knowledge management” [45] (p. 211). The combination of DIPs with knowledge management in general, or Semantic Web technologies in particular, is a promising approach.

**Author Contributions:** Conceptualization, R.J.P.; methodology, R.J.P.; software, R.R.P. and R.J.P.; validation, R.R.P. and R.J.P.; formal analysis, R.J.P.; investigation, R.J.P. and R.R.P.; resources, R.R.P. and R.J.P.; data curation, R.J.P. and R.R.P.; writing—original draft preparation, R.J.P. and R.R.P.; writing—review and editing, R.J.P. and R.R.P.; visualization, R.J.P. and R.R.P.; supervision, R.J.P.; project administration, R.J.P. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Not applicable.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Wagner, B.; Monk, E. *Enterprise Resource Planning*, 3rd ed.; Cengage Learning: Boston, MA, USA, 2008.
2. Data Integration. Available online: <https://dataintegration.info/data-integration> (accessed on 15 July 2021).
3. Zacharewicz, G.; Daclin, N.; Doumeingts, G.; Haidar, H. Model Driven Interoperability for System Engineering. *Modelling* **2020**, *1*, 94–121. [CrossRef]
4. Chen, D.; Daclin, N. Framework for Enterprise Interoperability. In Proceedings of the IFAC Workshop EI2N, Loria, France, 12 July 2006; pp. 77–88.
5. Petrasch, R. Data Integration Patterns in the Context of Enterprise Data Management. In *Recent Advances in Information and Communication Technology 2018, Proceedings of the 14th International Conference on Computing and Information Technology (IC2IT 2019), Bangkok, Thailand, 4–5 July 2019*; Unger, H., Sodsee, S., Meesad, P., Eds.; Springer: Berlin/Heidelberg, Germany, 2019.
6. Object Management Group; Model Driven Architecture (MDA). MDA Guide rev. 2.0, OMG Document ormsc/2014-06-01. 2014. Available online: <https://www.omg.org/cgi-bin/doc?ormsc/14-06-01> (accessed on 21 February 2022).
7. Petrasch, R.; Meimberg, O. *Model Driven Architecture—Eine Praxisorientierte Einführung in die MDA*; Dpunkt.Verlag: Heidelberg, Germany, 2006.
8. OMG (Object Management Group). OMG Unified Modeling Language (OMG UML), Version 2.5.1, Document Formal/2017-12-05. 2017. Available online: <https://www.omg.org/spec/UML/2.5.1/PDF> (accessed on 21 February 2022).
9. Shervin Ostadzadeh, S.; Shams Aliee, F.; Arash Ostadzadeh, S. An MDA-Based Generic Framework to Address Various Aspects of Enterprise Architecture. In *Advances in Computer and Information Sciences and Engineering, Proceedings of the 2007 International Conference on Systems, Computing Sciences and Software Engineering (SCSS), Bridgeport, CT, USA, 3–12 December 2007*; Springer: Berlin/Heidelberg, Germany, 2007.

10. OMG (Object Management Group). MOF Model to Text Transformation Language; Version 1.0, Document Formal/2008-01-16. January 2008. Available online: <https://www.omg.org/spec/MOFM2T/1.0/PDF> (accessed on 21 February 2022).
11. Eclipse Foundation. Obeo Aceleo Project Website. Available online: <https://www.eclipse.org/aceleo/> (accessed on 12 September 2021).
12. IBM. Rational Software Architect Documentation, Vers. 9.7: Specifying Stereotypes and Constraints for Custom UML Profiles. Available online: <https://www.ibm.com/docs/en/rational-soft-arch/9.7.0?topic=metamodel-specifying-stereotypes-constraints> (accessed on 15 December 2021).
13. Petrasch, R. DIP Github Repository. Available online: <https://github.com/rpetrasch/DIP> (accessed on 4 February 2022).
14. Hohpe, G.; Woolf, B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*; Addison-Wesley: Boston, MA, USA, 2012.
15. Apache Foundation. Apache Camel. Available online: <https://camel.apache.org> (accessed on 1 February 2019).
16. Halevy, A.; Doan, A.; Ives, Z. *Principles of Data Integration*; Elsevier/Morgan Kaufmann: Burlington, MA, USA, 2012.
17. Reeve, A. Managing Data in Motion: Data Integration Best Practice Techniques and Technologies. In *The Morgan Kaufmann Series on Business Intelligence*; Morgan Kaufmann: Burlington, MA, USA, 2013.
18. Mulesoft. Top Five Data Integration Patterns. Available online: <https://www.mulesoft.com/resources/esb/top-five-data-integration-patterns> (accessed on 12 December 2021).
19. Majkić, Z. *Big Data Integration Theory: Theory and Methods of Database Mappings*; Springer: Berlin/Heidelberg, Germany, 2014.
20. Rahm, E. Data Integration in the Life Sciences. In Proceedings of the First International Workshop (DILS 2004), Leipzig, Germany, 25–26 March 2004.
21. Blokdyk, G. *Data Integration Patterns a Complete Guide—2020 Edition*; 5STARCOoks: Plano, TX, USA, 2019.
22. Modelmapper Website. Available online: <http://modelmapper.org/>; <https://github.com/modelmapper/modelmapper> (accessed on 12 December 2021).
23. Jitterbit Website: Products “Harmony” and “B2B/EDI Integration”. Available online: <https://www.jitterbit.com/> (accessed on 12 December 2021).
24. ZadahmadJafarloua, M.; Moeninib, A.; YousefzadehFarda, P. *New Process: Pattern-Based Model Driven Architecture*; Procedia Technology; Elsevier: Burlington, MA, USA, 2012; Volume 1, pp. 426–433.
25. Hruby, P. *Model-Driven Design Using Business Patterns*; Springer: Berlin/Heidelberg, Germany, 2006.
26. Kendle, N. The Enterprise Data Model. 2005. Available online: <https://tdan.com/the-enterprise-data-model/5205> (accessed on 1 October 2021).
27. Richardson, C. *Microservices Patterns: With Examples in Java*, 1st ed.; Manning: Shelter Island, NY, USA, 2018.
28. Blaha, M. *Patterns of Data Modeling*; CRC Press: Boca Raton, FL, USA, 2010.
29. Lenzerini, M. Data integration: A theoretical perspective. In Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '02), Madison, WI, USA, 3–5 June 2002; pp. 233–246.
30. Evans, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*, 1st ed.; Addison-Wesley Professional: Boston, MA, USA, 2003.
31. Fowler, M. Bounded Context. 2014. Available online: <https://martinfowler.com/bliki/BoundedContext.html> (accessed on 13 September 2021).
32. Khononov, V. *What Is Domain-Driven Design?* O'Reilly Media Inc.: Sebastopol, CA, USA, 2019; Available online: <https://www.oreilly.com/library/view/what-is-domain-driven/9781492057802/ch04.html> (accessed on 2 September 2021).
33. Price, E.; Buck, A.; Kshirsagar, D.; Sherer, T.; Boeglin, A.; Stanford, D. Identifying Microservice Boundaries. Available online: <https://docs.microsoft.com/en-us/azure/architecture/microservices/model/microservice-boundaries> (accessed on 12 October 2021).
34. Sumathi, S.; Esakkirajan, S. *Fundamentals of Relational Database Management Systems*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2007.
35. Lee, R.C.; Tepfenhart, W.M. *Practical Object-Oriented Development with UML and Java*; Prentice Hall: Hoboken, NJ, USA, 2002.
36. Papazoglou, M.; Spaccapietra, S.; Tari, Z. *Advances in Object-oriented Data Modeling*; MIT Press: Cambridge, MA, USA, 2000.
37. Fowler, M. *Refactoring: Improving the Design of Existing Code*, 2nd ed.; Addison-Wesley Professional: Boston, MA, USA, 2018.
38. Gamma, E.; Helm, R.; Johnson, R.; Johnson, R.E.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley Professional Computing: Boston, MA, USA, 1994.
39. Ponniah, P. *Data Warehousing Fundamentals for IT Professionals*, 2nd ed.; Wiley: Hoboken, NJ, USA, 2010.
40. Kimball, R.; Caserta, J. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*, 1st ed.; Wiley: Hoboken, NJ, USA, 2004.
41. OMG (Object Management Group). Object Constraint Language (OMG OCL) Version 2.4, Document Formal/14-02-03. 2014. Available online: <https://www.omg.org/spec/OCL/2.4/PDF> (accessed on 21 February 2022).
42. Hohpe, G. Enterprise Integration Patterns. Available online: <https://www.enterpriseintegrationpatterns.com> (accessed on 21 August 2021).
43. OMG (Object Management Group). Model Driven Message Interoperability (MDMI). v2.0—Beta 1, Document Number: Dtc/2021-01-02. 2021. Available online: <https://www.omg.org/spec/MDMI/2.0/Beta1/PDF> (accessed on 21 February 2022).

44. OMG (Object Management Group). Meta Object Facility 2.0 Query/View/Transformation (QVT). Version 1.3, Document Number Formal/16-06-03. 2016. Available online: <https://www.omg.org/spec/QVT/1.3/PDF-QVT/> (accessed on 21 February 2022).
45. Li, Y.-F.; Zhang, H. Integrating software engineering data using semantic web technologies. In Proceedings of the 8th Working Conference on Mining Software Repositories, Honolulu, HI, USA, 21–22 May 2011; Van Deursen, A., Xie, T., Zimmermann, T., Eds.; Association for Computing Machinery: New York, NY, USA, 2011; pp. 211–214.