TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

TOMMI RATAMAA

GENERIC DATA TRANSFER OBJECT AND ENTITY MAPPING

Master of Science Thesis

Examiner: Prof. Tommi Mikkonen (TUT)
The examiner and the subject were approved by the Faculty of Computing and Electrical Engineering on 8 May 2013.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
RATAMAA, TOMMI: Generic Data Transfer Object and Entity Mapping
Master of Science Thesis, 62 pages, 26 Appendix pages
December 2013
Major: Software engineering
Examiners: Professor Tommi Mikkonen, M.Sc. Janne Sikiö
Keywords: DTO, ORM, JPA, Web application development, Java EE

Layered architecture in Java EE web applications is one example of a situation where parallel, non-matching class hierarchies need to be maintained. The mapping of Data Transfer Objects (DTO) and entities causes manual overhead, more code to maintain and the lack of automated solution may lead to architectural anti patterns. To avoid these problems and to streamline the coding process, this mapping process can be supported and partially automated.

To access the problem, the solutions and related techniques to the mapping process are analyzed. For further analysis, a runtime mapping component approach is chosen. There are multiple techniques for mapping the class hierarchies, such as XML, annotations, APIs or Domain-Specific Languages. Mapping components use reflection for mapping but for actual copying of the values, dynamic code generation and caches can be used for better performance.

In this thesis, a comprehensive Business Process Readiness (BRR) analysis was performed. Analyzed categories included features, usability, quality, performance, scalability, support and documentation. The requirements for a generic purpose mapping component were derived from the needs of Dicode Ltd. Out of the eleven found implementations, six were chosen for the complete analysis based on feature category.

Finally, a rating in range from 1 to 5 was assigned to each of the components as a weighted average of the results in each category. There are notable differences related to usability, measured as the amount configuration needed, between the implementations. Additionally, components using dynamic code generation perform better compared to others but no scalability concerns were noted for a real application. Overall, based on the analysis, we found that there exists very good solutions to support the mapping process for Dicode Ltd. that can be recommended to be used in future projects.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
RATAMAA, TOMMI: Yleiskäyttöinen muunnos tiedonvälitysolioiden ja entiteettien välillä
Diplomityö, 62 sivua, 26 liitesivua
Joulukuu 2013
Pääaine: Ohjelmistotuotanto
Tarkastajat: professori Tommi Mikkonen, DI. Janne Sikiö
Avainsanat: DTO, ORM, JPA, Web-sovelluskehitys, Java EE

Rinnakkaisia, toisistaan rakenteeltaan poikkeavia luokkahierarkioita tarvitaan muun muassa kerrosarkkitehtuurilla toteutetuissa Java EE -pohjaisissa websovelluksissa. Tiedonvälitysolioiden (engl. Data Transfer Object) ja entiteettien välinen muunnos aiheuttaa manuaalista työtä ohjelmoijalle, lisää ylläpidettävää koodia ja toisaalta automatisoidun ratkaisun puuttuminen voi johtaa arkkitehtuurin kannalta haitallisiin piirteisiin. Näiden haasteiden välttämiseksi tämä muunnosprosessi on osittain automatisoitavissa.

Tekniset ratkaisut ongelman ratkaisemiseksi analysoitiin ja tarkempaan käsittelyyn valittiin lähestymistapa, jossa muunnos suoritetaan ajonaikaisesti. Luokkahierarkioiden rakenteen kohdentamiseen voidaan käyttää useita eri tekniikoita, kuten XML:ää, annotaatioita, ohjelmointirajapintoja tai toimialueeseen sidonnaisia kieliä (engl. Domain-Specific Language). Kohdentamisessa käytetään Javan reflektointia mutta varsinaiseen arvojen kopiointiin voidaan saavutettujen tehokkuusetujen vuoksi hyödyntää ajon aikana tuotettua ohjelmakoodia sekä välimuisteja.

Toteutusten vertailuun käytetään Business Process Readiness -arviointia, josta on käytössä toiminnallisuuden, käytettävyyden, laadun, tehokkuuden, skaalautuvuuden, tuen ja dokumentaation osa-alueet. Toiminnalliset vaatimukset on johdettu Dicode Oy:n tarpeista. Näiden pohjalta yhteensä yhdestätoista arvioidusta toteutuksesta kuusi valittiin kattavamman arvioinnin vaiheeseen, jossa kokonaisarvio muodostui kaikkien arvioitujen osa-alueiden painotetusta keskiarvosta välille 1-5.

Käytettävyyttä mitattiin vaaditun konfiguraation määrällä, ja tällä osa-alueella toteutusten välillä havaittiin merkittäviä eroja. Ajon aikana ohjelmakoodia tuottavat toteutukset erottuivat tehokkuusmittauksista, mutta todellisen sovelluksen tapauksessa mitattavissa olevia skaalautuvuuseroja ei havaittu. Vertailun pohjalta voidaan todeta, että Dicode Oy:n tarpeisiin on olemassa erittäin hyviä toteutuksia ja niiden käyttöä voidaan suositella tulevissa projekteissa.

PREFACE

One of the main reasons why I like doing software is that possibilities and innovations can only be constrained by limitations derived from real world. As it happens, this thesis work was actually initiated as a result of a powerful and many times challenging phrase 'It just can't be done' left in the air in an internal development meeting at Dicode in late 2011 after I had just asked: 'Couldn't this be automated?' The current topic was the conversion between DTO and entity objects that we all were so bored having to do. As a result, I wrote first version of the Generic DTO Converter and released it in late December the same year.

The second motivation was the notation of multiple implementation with most being developed around 2011 and 2012 with still no publications on the topic apart from benchmarks, blog entries and debates over if DTO pattern should be used or not. Doing this thesis work has largely enlightened me on the topic and pointed a lot of places for improvement in my implementation. The reason behind me authoring an open-source component and writing this thesis in English is the will to share this information to and ease the pain of the rest of the developers trying to tackle the same problem.

I want to thank my colleges at Dicode for great support and suggestions for improvement. Especially, I want to thank my supervisor, Janne Sikiö and my examiner, professor Tommi Mikkonen.

At Tampere 10th Dec 2013

Tommi Ratamaa
Atomikatu 3 B 43
33720 TAMPERE
+358 45 123 9353
tommi.ratamaa@gmail.com

CONTENTS

LIST OF ABBREVIATIONS

API	Application Programming Interface
BRR	Business Readiness Rating
DAO	Data Access Object
DSL	Domain-specific Language
DTO	Data Transfer Object
EJB	Enterprise JavaBeans
ESB	Enterprise Service Bus
GPL	General Purpose Language
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IoC	Inversion of Control -pattern
Java EE	Java Enterprise Edition
JAXB	Java Architecture for XML Binding
JDBC	Java Database Connectivity
JIT	Just-In-Time
JPA	Java Persistence API
JRE	Java Runtime Environment
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LGPL	GNU Lesser General Public License
MVC	Model-View-Controller
OpenBRR	Business Readiness Rating for Open Source

ORM	Object-Relation-Mapping
PDF	Portable Document Format
POJO	Plain Old Java Object
RMI	Remote Method Invocation
SOA	Service Oriented Architecture
UI	User-Interface
XML	Extensible Markup Language

1. INTRODUCTION

Some software architecture related decisions, such as the use of server-client architecture or layered architecture, introduce a requirement for maintaining multiple class hierarchies that usually have similarities but are used for different purposes. For the client object hierarchies usually reflect the presentation whereas server side hierarchies reflect the underlying data model or database schema.

In its most generic level a web application consists of a client, a server and usually a database. In this thesis the focus lies on the server side of a Java EE web application. More specifically, we focus on mapping the class hierarchies between the layers within the application. Mapping the object hierarchies between these two layers is usually a straightforward programming task. However, doing it manually consumes time that could be spent in more productive tasks, creates more code to be tested and maintained, reduces flexibility, introduces new bugs or leads to architectural anti-patterns when avoided.

In this thesis the aim is to find a general technical solution to support the mapping of object hierarchies between the architecture layers of Java EE web applications. This solution is targeted for the needs of software company Dicode Ltd. The ideal solution would be practical to use, minimize the amount of manual work and cover most of the typical use scenarios while still maintaining high customizability. The solution should also maintain relative efficiency and scalability.

In Chapter 2, the problem domain and environment are introduced. Different use scenarios are analyzed. In Chapter 3, technical solutions, implementation techniques available and their possible limitations or challenges are analyzed. In Chapter 4, the requirements for the solutions are described and derived from practice to match the need of Dicode Ltd. In Chapter 5 the different solutions are compared based on a comprehensive maturity model. Finally, in Chapter 6 the results of the comparison are concluded.

2. ENVIRONMENT AND USE CASES

In order to specify the requirements related to the different use cases for a generic Data Transfer Object (DTO) and Entity mapping solution, we first need to analyze the usage environment and related patterns. In Section 2.1 we will first look at the general architecture of a Java EE web application and the used frameworks in Section 2.2.

After describing the environment, we will more specifically determine the different patterns and technologies related to the usage of Entity and DTO classes in Sections 2.3 and 2.4. Finally, we will look at the most typical use cases for DTO and Entity mapping in Section 2.5 and other possible use cases in Section 2.6.

2.1 Java EE Web Application Architecture

A typical Java EE web application server is based on a multi-tier architecture where components on a certain tier may only access the components of the same or lower layer and shall usually not skip the layers in between. While different applications of the common the Model-View-Controller (MVC) pattern are used in various web frameworks, such as Ruby On Rails, a Java EE setting typically consists of View, Controller, Service and Data Access Object (DAO) tiers (see Figure 2.1). [1]

Views are responsible for representing the Graphical User Interface (GUI) and passing the user actions to Controllers. Controllers on the other hand retrieve the data from Services to be presented in Views and handle the user input and pass them to Service-level APIs for further processing. Services are responsible for the business logic related to different actions, processing the data, transactions and data retrieval from the DAO layer. DAO level is responsible for creating an abstraction over the actual data storage implementation, typically a database management system. DAO level APIs use entity classes which present the data of the underlying data storage typically working as an Object Relation Mapping (ORM) [2] style meta model of the data.

Views and Controllers are usually closely coupled and created for a specific purpose or an user interaction with specific information needs, whereas Services work on a richer data model, i.e. entities, representing the business logic of the application. Retrieval of the entities, filtering and mapping of them to meet the

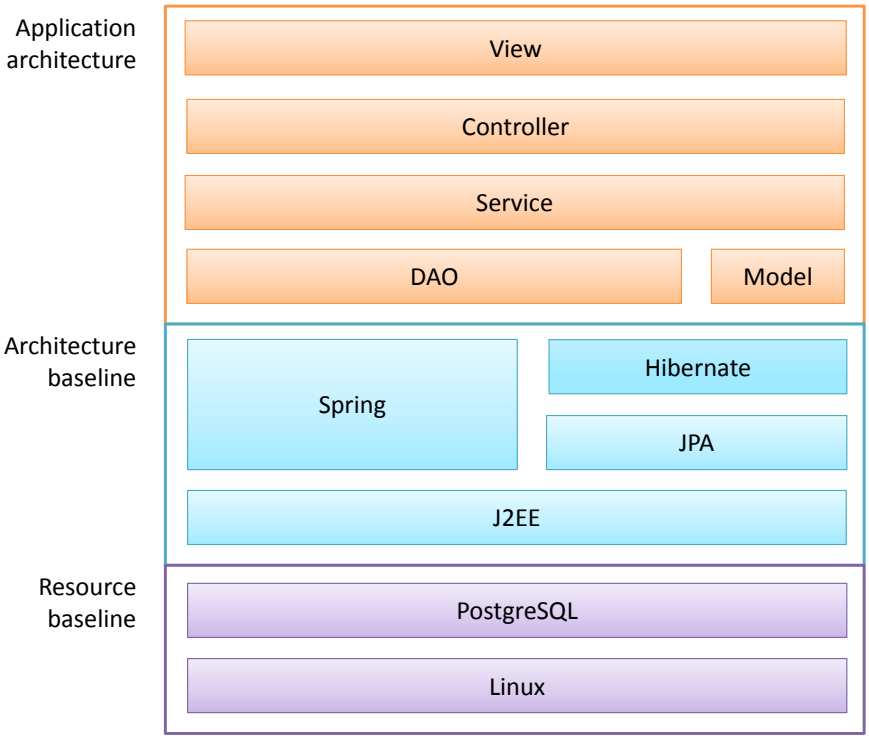


Figure 2.1: A software architecture of a Java EE application using Spring and Hibernate.

specific information needs of the Controller is related to the business logic and therefore the responsibility of the service level. Service level APIs are thus designed to match with the information needs of Controllers. This is done by the use of DTOs instead of lower level Entities in the service level APIs, which sets DTO and Entity mapping to be the responsibility of the service level.

Usually in Java EE applications the transaction scope of the application is limited to service level. This makes perfect sense since this tier should be the only tier where DAO methods resulting in database queries are to be called. Also the decision to commit or roll back a transaction as well as the related error handling is depended on the business logic and rules set to the domain model.

2.2 Java EE Application Frameworks

In a Java EE web application server individual component objects are referred to as beans. A Java EE web application is a multi-threaded environment where sharing of resources between requests, user sessions and the whole application is controlled by the scopes of the beans. Beans may have different scopes and there may be multiple simultaneously existing implementations for the same components in different scopes. In Spring framework scopes may be restricted to e.g. the whole

application, a single user session or a single page request [3]. Typically controllers are request or session scoped meaning that there may be as many duplicate components as there are concurrent users for the application. On the other hand services and DAOs are usually application scoped singleton beans that may be simultaneously called from multiple Controllers.

A web application framework is responsible for the construction and initialization of the beans and handling of the appropriate scopes and references between the beans. This requires the use of inversion of control pattern to handle inter-dependencies between the beans. Typically this is achieved by dependency injection pattern using either setter or field injection with interface injection pattern. Java EE frameworks, such as Spring [1] and JBoss Seam [4], usually also provide a number of additional features such as handling of transactions, exceptions, security aspects and so on by creating a middle-layer between bean method invocations utilizing dynamic proxies or dynamic code generation [1].

2.3 Java Persistence API and Entities

Java Persistence API (JPA) [5] provides a standard way to do Object Relational Mapping (ORM) in Java EE applications [2]. JPA specifies the entity mappings between database schema and the entity model in both Extensible Markup Language (XML) and Java annotation formats as well as an abstract query language tied to these entity mappings [5]. The actual database layer abstraction itself in JPA implementations is based on traditional Java Database Connectivity (JDBC) drivers but JPA provides the industry standard of accessing these APIs in an object oriented ORM manner.

As of today, multiple implementations of the JPA version 2 specification exist, such as JBoss Hibernate [6], Oracle Toplink [7] or EclipseLink [8], Batoo JPA [9] and Apache OpenJPA [10]. With the use of JPA the Java EE application is not directly depended on any particular persistence implementation, which enables higher interoperability. There also exists a vast tooling support for example for generating entity classes from a database schema and vise versa.

Entities provide the way of reading, creating, updating and deleting database data through directly accessing the entity objects. In practice, the entity classes are simple Java beans for which the JPA implementation creates dynamic proxies for. Modifications to persistent entities is automatically populated to database while the relational entity mappings could be set to e.g. lazily fetch related entities when reading the data sets properties of an entity object and cascading the deletions or additions to these sets where necessary.

Traditionally the entity mappings have been done in separate XML files loosely

related to the actual entity classes but as of annotation support introduced in Java version 1.5 [11], annotation mappings have become more and more popular. Both XML and annotation mappings provide the means of mapping database column, their types, type conversions, primary keys, table relations, inheritance, sequence generation and some of the database layer constraints [5]. Initially, these mappings are usually generated with tools while the differences between these mapping techniques become almost irrelevant. However, when the application is in maintenance state, the level of coupling becomes more important which tends to support the choice of annotation mapping over XML mappings.

2.4 Data Transfer Object Pattern

DTO pattern is often used when two remote processes communicate with each other to limit the communication overhead and network latency to one single call instead of many targeted to a remote object. DTOs are usually simple, hierarchical data containers with getters and setters but with no actual logic. They include all the necessary data for one use scenario usually flattering the structure of more complex domain entities. If anything, they should only be responsible for serializing and deserializing themselves to other formats, such as XML. Assembler or Mapper pattern is often used to transform domain entities or the remote objects to DTOs and vise versa in order to avoid dependencies between DTO and entity classes, since the classes should be serializable between system boundaries. [12; 13]

There are, however, justified use cases for the DTO pattern outside remote call interfaces. DTOs are often used in a multi-tier architecture in APIs between the business and presentation layers [14], just as in our Java EE architecture described earlier. Actually, in this context domain entities might as well be thought as remote objects since calls to their methods can cause lazy database queries, which in fact crosses system boundaries.

The only simpler alternative solution for the usage of DTOs in service level APIs would be passing entity models to views directly and vise versa. This, however, creates multiple architectural concerns. These include tight coupling of views, controllers and the business logic, bypassing the responsibility of the service tier by giving the presentation direct access to domain entities, possibly problematic and hard to customize data structure for the use scenario in hand and possible side effects when accessing data related to entity objects in views. In fact, Dino Esposito claims that the only drawback of using DTOs is the additional work related to creating and managing them [14], which is even claimed to be impossible to generate automatically [12].

The loose coupling between views and the service level is important for the sake

of both abstraction provided by the service interfaces and for the ability to have multiple or replaceable view implementations using the same services. This loose coupling allows the business model to evolve over time without directly affecting all the views.

Service implementations might also change over time and there could be a requirement to use external web services instead of a local database as the application scales or environment changes. In this scenario the data might even no longer be represented by entity classes but due the usage of DTOs classes the service API may remain intact. Also, whereas entities are tied to the system they are used in, DTOs may easily be serialized and deserialized over system boundaries [14], and could thus be used directly in for example web service calls, Remote Method Invocations or in a Service Oriented Architecture (SOA) setting.

Additionally, with most entity frameworks, there are also side effects while accessing the properties of entity objects. The entity objects are usually actually proxies and the simple getters could actually fire lazy database queries and thus start multiple transactions in the background of processing the view. What is worse, being not tied to an entity manager or a session provided by the entity framework, these results may not be cached and thus could result in multiple queries as the getters could be called multiple times during the rendering of a view. This could have severe negative effects to the performance of the application.

As an attempt to fix these kind of issues the entity session could be kept open during the view processing phase, but entity frameworks also automatically update the existing data in the database while the entity is bound to a session or a manager. Thus, modifying the entity in the view or controller, may then lead to bypassing the service level, the related transaction handling, the business logic related actions and validations or even security related aspects entirely. This would basically mean that entities would provide an API outside of the service level for modifying all the related data in the underlying database, data that in many cases should not be modified by the user at all.

2.5 DTO Usage Scenarios

DTOs are used to represent the rich entity model for a specific purpose. These purposes could be various and not all of the usage scenarios of DTOs can be known beforehand nor analyzed but by looking at the most typical ones based on the experiences in previous projects, we could access the most important features and requirements for the mapping component.

The usage of DTOs is not limited to presenting data in a HyperText Markup Language (HTML) rendered GUI alone but also extends to other representations

created using service level APIs, such as data transferred in a JavaScript Object Notation (JSON) interface, XML files generated for and read from external systems, Portable Document Format (PDF) or Excel reports or emails. It is not important, however, to consider these representations separately for the structure of the data and the feature requirements related are essentially the same.

In the past projects most typically DTOs have been used for presenting sub-portions of the data of an entity in a specific report, viewing the data related to an entity aggregation group, editing the whole entity aggregation group at once, showing a simple listing of certain entities for selection. DTOs are also used in more complex listings and reports. Notably the structure of the DTO classes, the properties present as well as their names and data types might be different in each of these usage scenarios.

2.5.1 Creating, Viewing and Modifying Entity Groups

Entities reflect the database columns and thus the business model of the application. In this business model, certain groups of entities are more highly coupled, forming aggregation entity groups. Such groups are usually edited, viewed and modified as a whole. A simple example of such an entity group could be an order related to a certain customer with a special delivery address and order items related to products (see Figure 2.2).

This kind of structure might as well be represented in DTO classes with similar structure as seen in Figure 2.3 for editing purpose. Notably in this setting, however, part of the information in the richer entity model is not present and only references to the primary keys of customer and product are used. In practice, a `OrderEditDto` would contain a `java.util.List` of `OrderItemEditDtos`.

When fetched for editing, typically the Order entity would be fetched by its primary key and there could be one actual query for the `Order`, one for the delivery `Address` and one for the related `OrderItem`s. With eager fetching mode, this could be reduced to two queries, but since this kind of view would only require these entities, the significance in performance would most likely be irrelevant.

For creation and modification purposes, the foreign key references of such related entities that need not to be edited are stored in DTO classes. When the modifications are saved, these related entities are fetched by their primary keys when necessary. In this example we only have one to many and many to one relational mappings. For many to many mappings lists or arrays of related entities' primary keys are usually stored instead.

Similar aggregation structure in DTO classes could be present for the viewing of this entity aggregation group (see Figure 2.4). However, in this case `OrderViewDto`

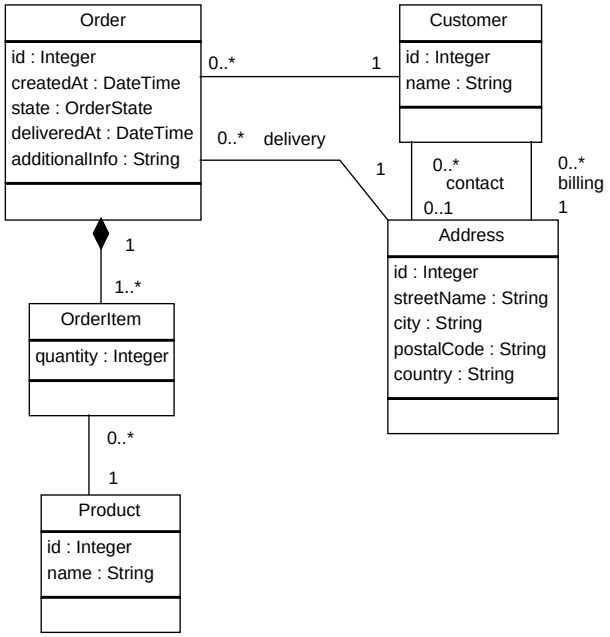


Figure 2.2: A simple entity aggregation group.

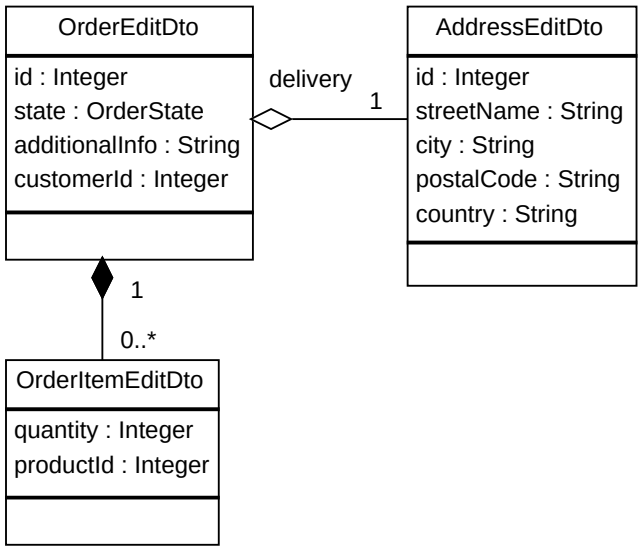


Figure 2.3: DTO presentation of the entity group for editing.

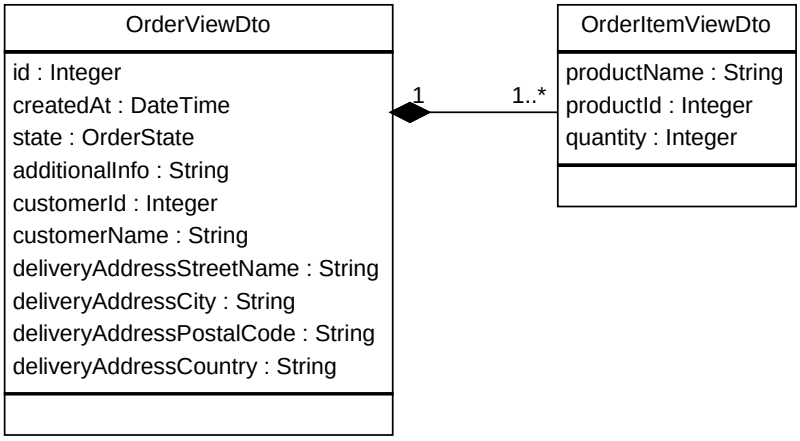


Figure 2.4: DTO structure for viewing the entity group.

also contains the name property of `Customer` and `OrderViewDto` contains the name of the related `Product`.

Notably, the structure is now also flatterned so that `OrderViewDto` contains the fields of `Address`. Again, assuming that only details of one order are viewed at once, lazy queries performed by the property getters of JPA entities could be efficient enough.

2.5.2 Listings and Reports

The use scenario of DTOs for listing or reporting purposes compared to viewing or editing a single entity group differ in their performance requirements due to the number of entities fetched. Additional queries for each fetched entity, O(N), should be avoided. Thus accessing related data, such as the customer related to an order or an address related to the customer, through entity properties is not usually possible due the lazy O(N) queries related.

In some scenarios, though, O(N) queries could be avoided by the use of eager fetching mode in entity mappings and the use of JPA query language or criteria API. In such scenarios JPA implementations would do one query for the result group fetching the primary keys of each ids and an additional query for each related entity property. Hence the number of queries performed would be O(1). However, since in most cases there exists other use scenarios for the same entities where eager fetching is not desired, other means are necessary.

The related data could be included in the result sets of these database queries by either selecting all the necessary entities in a container object or mapping query results to a DTO class directly. The former is usually easier since JPA implementations can automatically generate the listing of selected columns and map

result sets to entity objects with appropriate data type conversions in place whereas with direct DTO mapping property by property mapping and explicit data type specification is usually necessary. On the other hand, fetching all related entities may result in more than necessary data to be fetched for the following DTO mapping. For example, if only customer name if needed from customer entity, all the other fields fetched are waste of database, network and memory resources.

In some cases, however, reports and listings may also contain summary or aggregate information over larger data sets that can not be calculated efficiently by fetching all the data from the database. In these cases only explicit mapping of query results and DTO classes can be used. An example of such a report could be a listing of the monthly total order quantities of each customer for certain products.

2.5.3 Findings From Previous Projects

Over the past few years, DTO pattern has been successfully utilized in many Java EE projects having a multi-tier architecture at Dicode. We analyzed five of these projects of different sizes and compared their usage of DTOs in relation to their size in source files and entity model. The findings are presented in Table 2.1 where Classes represent the total number of classes or interfaces in the project, entities describe the total number of entity classes, DTOs the total number of DTO classes, DTO-% the relational number of DTO classes in the total, DTOs/Entities the relation between DTO and entity classes and DAO DTO-% the relational number of DAO level DTO classes of all DTO classes.

Project	**Classes**	**Entities**	**DTOs**	DTOs-%	**DTOs/ Entities**	**DAO DTO-%**
A	1053	104	180	17 %	1.8	5 %
B	1101	68	163	15 %	2.3	3 %
C	554	32	76	14 %	2.4	0 %
D	372	50	40	11 %	0.8	5 %
E	140	11	20	14 %	1.8	0 %
Average				14.2 %	1.8	2.6 %

Table 2.1: DTO usage in some of the past projects.

The findings suggest that the number of DTO classes used tends to be relative to the size of the project with DTO classes representing roughly one 7th of all the classes in the project. With project D as an exception, there also seems to be an expected relation between the number of entity classes and DTO classes: for each

entity there exists on average two DTO classes. Also it can be concluded that DAO level DTOs, related to DTO classes directly selected in special database queries, represent only a minor portion (less than 3 % on average) of all the DTO classes. In terms of simplifying the mapping process, Query and DTO mapping should therefore not be considered as important as entity and DTO mapping.

As far as maintainability is concerned, it would be optimal to have separate DTO classes for each use cases such as viewing, editing and listing a certain entity. This would allow flexibility in the future so that any of these use cases could be independently altered according to future needs. Considering this, the DTO/Entity-relation figure is relatively low, taken that most of the entities with some exceptions, are usually at least viewed and edited. This suggests, and by further analysis could be verified, that DTO classes are being reused for multiple purposes.

One cause for this anti-pattern is identified to be the amount of manual work related to DTO and entity mapping, tempting a developer to take the easier way. These kind of mapping tasks are often treated as boring and repetitive, taking developers time from the productive work. They can usually involve mapping the fields in both directions and having separate methods for collection conversions.

Project A was also further analyzed in order to determine whether DTO conversions are done in assembler or mapper components that the DTO pattern suggests. As there should be only single responsibility for each class for maintainability, and because the same entities and DTOs could be used in different services, there is a general architecture rule to place the DTO and entity mappings related to an entity aggregation group to a specific converter component. However, the findings from this analysis show that nearly half (47 %) of the DTO and entity mappings was actually done inside service implementations. This is another concern related to the overhead of doing DTO and entity mappings manually.

2.5.4 The Scope of DTO and Entity Mapping

For the different analyzed database related DTO use scenarios, there exist different requirements for both data structure and application performance. Especially the performance requirements specify both the DAO tier query result structure as well as the tier in which DTOs are built as shown in Table 2.2.

Out of these scenarios, JPA implementations provide some support for query result to DTO mapping in forms of query result transformers which are able to bind selected query columns or aliases to bean properties [5]. However, this usually requires the programmer to specify these mappings with the correct type mapping explicitly in code. These operations could be further automated by query builders

Scenario	**Mapping tier**	**Query result**	**Mappings**
Create	Service	N/A	DTO → *Entity*
View	Service	Entity	Entity → *DTO*
Modify	Service	Entity	Entity → *DTO* → *Entity*
Simple listing	Service	Entity list	Entity → *DTO*
Complex listing	Service	Entity container list	Entity container → *DTO*
Complex listing	DAO	DTO list	Query result → *DTO*
Aggregate report	DAO	DTO / DTO list	Query result → *DTO*

Table 2.2: DTO and entity mapping tier.

which hold the type and alias name information of the selected columns or aggregate data. These kind of queries are, however, usually related to the most complex minority of queries in the application and thus usually highly customized.

In most of the scenarios the mapping could be accessed with a DTO to entity or entity to DTO mapping in the service tier of the application. Complex listings may alternatively be mapped from entity container lists or query results directly depending on the performance requirement in question.

2.6 Other Generic Use Cases

Given that the mapping of DTO and entity classes generalizes to the problem of mapping any two Java classes with each other, other usage for this kind of mapping component can also be found. Two of these use cases where a generic purpose DTO and entity mapper component has been successfully utilized at Dicode for other than a mapping between a DTO and an entity, are presented in this section. In these cases DTO and JAXB classes and two different entity classes are mapped with each other. Additionally, such a component could also be used in an easily customized deep cloning of an object where a class would be mapped with itself.

2.6.1 XML Class and DTO Mapping

In Java EE applications, Web Services are commonly used to communicate with external parties or to transfer data between system components in a SOA architecture. In Web Services the data transferred is represented in XML format. XML is also popular format for example for storing application settings or other data records such as financial information. Typically, the data is mapped automatically

with an object model using techniques such as Java Architecture for XML Binding (JAXB) [15]. Moreover, JAXB classes are usually automatically generated from XML schema or Web Service descriptions.

Often the schemes for XML files or Web Service descriptions are provided by external parties and might be subject to change. This means that whenever, for example, an additional element or attribute is added to the scheme, the automatically generated JAXB classes need to be regenerated. Basically, this makes it impossible to make manual changes to these classes and discourages the direct use of them in Service level interfaces due to easily spreading implications of their uncontrolled modifications. Thus, although JAXB objects are DTOs themselves, separate DTO classes might be used to provide stability to the system.

The structure of JAXB classes is essentially the same as that of a complex DTO class. It is an hierarchical aggregation containing fields with basic data types or single or ordered collections of elements. JAXB classes could be mapped with custom DTOs if there exists a need for them in Service interfaces or they could be mapped with entities directly. With the mappings specified in DTO or entity classes or a separate mapping file, the code dependencies to schema dependent JAXB classes could be close to zero and thus modifications to the application upon schema change would be minimal or even none for new optional elements or attributes.

JAXB generated classes also use some custom types targeted for XML documents specified in `javax.xml.datatype` package, such as `XMLGregorianCalendar`[15], that are typically not used elsewhere in the application. The JAXB definition also allows the use of public fields instead getter/setter methods specified by the bean standard. Typically, in the generated classes arrays as are used instead of `java.util.Collection` classes. Basically, these kind of differences require extra manual work for the conversion and make JAXB classes impractical to be used for anything other than XML mapping also encouraging the use of separate DTOs. With the use of the generic mapper component the inconsistencies in data types and the usage of fields could also be easily overcome.

2.6.2 Data Model Migrations

Sometimes data must be exchanged between two databases with different schemes. These kind of scenarios are typical when a system replaces and earlier system. During a certain period, these systems might also run in parallel and there could be a need for two-way synchronization of data. When working with JPA, the easiest solution is to generate entity models for both databases and map data between them.

In this setting, the mapping would occur between an entity class and other entity class. There would most certainly exists cases where a custom conversion should be

implemented due to the non matching schemes but many of the mapping would be similar to DTO and entity mapping where matching properties are mapped with a possible name and data type and aggregate structure changes.

One of the most repeating general concern in matching two different databases schemes is, however, the mapping of foreign key relations between these two databases where the primary keys do not match with each other. However, while a generic DTO and entity mapper component provided the possibility to define custom converter between two data types, this problem could be generally solved by storing the primary keys of the other database and defining conversion between two interfaces, implemented by matched entities, each of which provided the data type and primary key of the other database.

3. IMPLEMENTATION TECHNIQUES

The implementation techniques used in a generic DTO and entity mapping are presented in this chapter. First, in Section 3.1 we will present an overall mapping process with variations derived from existing implementations and discuss the benefits and disadvantages related to each variation. After that, in Section 3.2 the different mapping techniques are presented.

At the end part of the chapter, in Section 3.3 we will focus more closely on reflection capabilities in the Java programming language as well as its performance. The use of caches and the related patterns are discussed in Section 3.4. Finally, dynamic code generation for gaining performance advantage is covered in Section 3.5.

3.1 The overall mapping process

The different strategies for the overall process is described in Figure 3.1. It can be divided into different stages: coding, compile time and runtime. There exists three different strategies with which the mapping process can be supported. A plugin for an Integrated Development Environment (IDE) can help a developer to generate the mapping source code at the coding stage. A compile automation plugin, on the other hand, could process the mappings on compile time. Lastly, a mapping component bundled with the software can either use reflection or generate code to perform the mapping in runtime. A combination of IDE plugin and a runtime component or a compile automation plugin would also be possible. In such a scenario the IDE plugin could be used to generate the mapping configuration to be used in later phase (see Figure 3.1: path 1, 3, 6).

The coding phase includes analysis of the classes being mapped, which is traditionally done manually (see Figure 3.1: path 2, 5, 4, 12). It could, however, be supported by an Integrated Development Environment (IDE) plugin which would use code time compiling and reflection of existing source code by providing mapping options for the developer (see Figure 3.1: path 1, 3, 4, 5, 4, 12). An IDE plugin could be used to generate the source code or a mapping configuration file (see Figure 3.1: path 1, 3, 6). Both can then be edited manually. However, an IDE plugin generating source code will most probably not be able to alter the source files after they have

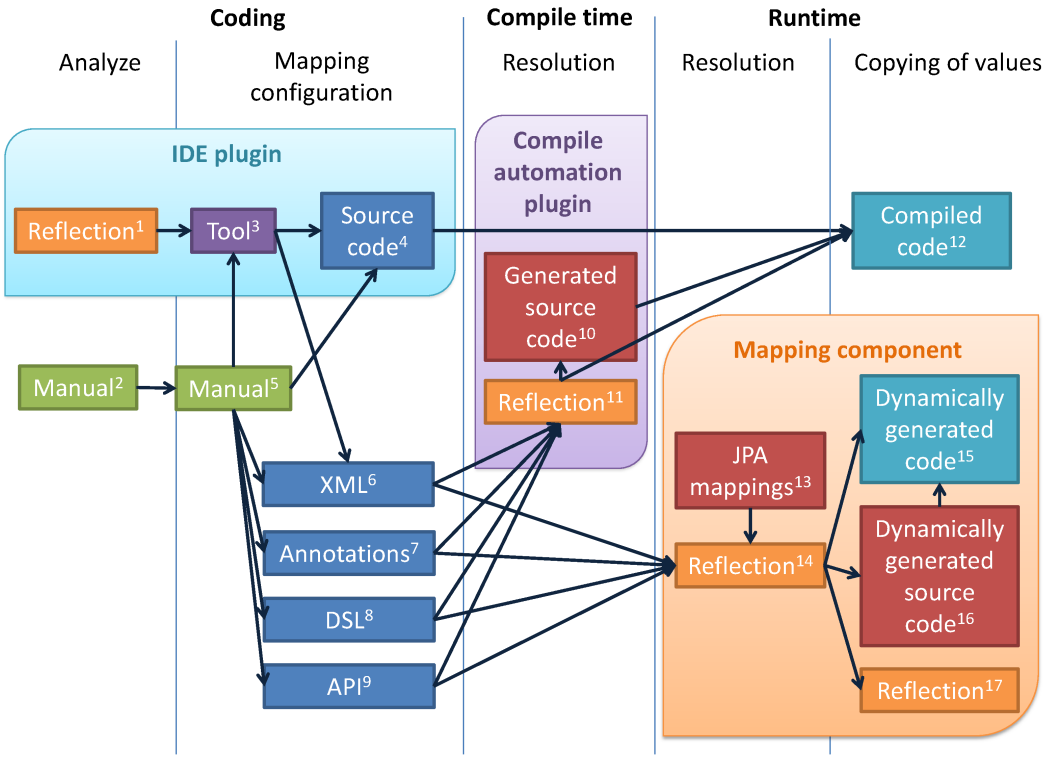


Figure 3.1: The process of mapping DTO and entity classes.

been manually edited, meaning that the support would only be partial compared to the use of a configuration.

While IDE plugins and Compile automation plugins are presented here mainly as theoretical approaches to automate DTO and entity mapping for completeness, the majority of existing implementations are actually, in fact, runtime mapping components. There exists at least one pure Eclipse IDE plugin based tool, Modelbridge [16], which, unfortunately however, during the writing of this thesis was unavailable for the at least two newest major versions of Eclipse. Some runtime components may, however, provide additional IDE plugins. Pure IDE plugins and Compile automation plugins alike would be highly dependent on the IDE or build automation tool used, sometimes even on its version, thereby reducing their generality in this problem domain. They would also not be able to access runtime information, such as JPA mappings, and are thus, could only provide a partial solution to the problem. They not focused further in this thesis.

There are several options available for the techniques used to define the mapping configuration (see Figure 3.1: path 6, 7, 8, 9) which are explained in more details in the next section. At some point, all of the approaches, except for the traditional manual one, need to access the properties and data types of the classes being mapped. This is highly based on the reflection capabilities provided by the Java

Runtime Environment (JRE). When it finally comes to the phase of actual copying of the values either reflection, code compiled from generated source code or dynamic code generation can be used which is explained more closely in Section 3.5. These are addressed more closely in the following sections.

3.2 Mapping technologies

Based on the analysis of existing implementations of mapping components the mapping techniques used to define mappings between DTO and entity classes in Java programming language include source code mapping done by using proxy objects or other APIs, XML based configurations and annotation based mapping configurations. It would also be possible to define the mappings using a Domain-specific Language (DSL).

All of the mapping techniques have their advantages and disadvantages. While, for example, XML mappings can be be more easily used with tools such as IDE plugins, they can easily generate overhead to the coding phase especially if no such tools exists. Annotation based mappings exists directly in the edited code and therefore could have benefits over other options, especially when it comes to maintenance. Mapping by proxy objects makes the mappings type and modification save but, on the other, makes it difficult to add meta-data to the configuration. DSL could be more expressive than XML or annotation mappings and may also include some complicated custom logic which could otherwise only be expressed in source code. Mapping techniques, however, are not exclusive and a single mapping component could support multiple techniques and let user choose the most appropriate.

3.2.1 XML

XML [17] has traditionally, among its countless other use purposes, been the most popular format for storing persistent configuration data in Java EE applications. Typed XML elements may contain attributes with standard or custom types. Elements may have children and form recursive structures. This structure is described in a schema file. The elements and attributes have a name-space, may contain attributes and children from different name-spaces or even different schemes. These schemes may refer to each other and multiple schemes may exist in a single document. This makes the format both flexible, easily extendable and, at the same time, with schema validation, formal enough to fit for almost any presentation of structural concepts.

When it comes to application configuration data, choosing XML format can be easily justified since it makes the configuration clearly separated from the application

logic. Because of the schema this configuration can be formally validated both in and outside of the application. There are also a great number of tools and technologies that help to manipulate, transform, or present XML data. With JRE provided APIs, such as the modern JAXB [15] implementation, it is also relatively straightforward to read, to modify and to perform schema validation for data stored in XML format with Java programming language. With JAXB, the XML element types can be mapped directly to Java classes and the elements may be read and modified type safely using these objects.

3.2.2 Annotations

Annotations were introduced in Java version 1.5 [11] for the purpose of providing meta-data in source code to related code elements. This meta-data can then be used by the compiler, software tools or frameworks using meta-programming. Annotations have different retention policies for these different use cases: Annotations with source-policy are removed at compile time while class and runtime policy annotations remain in the compiled binary class file but only annotations with runtime retention policy can be read at runtime using reflection. [18]

Annotations may have attributes which can be either required or have a default value. Annotation types themselves are specified as interface-like Java types, methods of which specifying the attributes. The data types for annotation attributes must be either primitive types or their wrappers types, java.lang.Strings, enumerations, `Class`-objects, other annotations or arrays of these. Null-values are not allowed even as default values but it is often possible to circumvent this by defining the type as an array with an empty default value or String with empty default value. [18]

Annotations may be placed in the source code before target elements that are allowed by the target specification of the annotation type. It may include any combination of elements that are defined as `java.lang.reflect.AnnotatedElement`s, including fields, methods, parameters, constructors, local variables, packages and other annotations. Only one annotation for a certain annotation type can be specified for each source code element [18]. However, cases where multiple annotations are needed can often be constructed by the use of an additional container annotation with a single array value attribute holding these annotations. Annotations can not be inherited but usually user specified annotations, that extends the behavior of a framework based on its own configuration annotations, are created by allowing annotation type as one of target elements of the annotation and reflecting these one step further.

Certain more hierarchical structures can be presented with annotations but, as

a considerable limitation, annotation types used in attributes can not directly or indirectly refer to the original annotation type. In other words, a recursive structure with a circular annotation reference is prohibited by the compiler. In practice, this means that annotations cannot be used to present a recursive structure with the complexity comparable to for example XML documents where an element node can contain instances of the same element node.

The limitations of recursive structures and inheritance are most probably related to the internal implementation of annotations which is basically a special kind of interface, which leads to the limitations in the inheritance hierarchy of Java. In fact, even though this was not intended as indicated with a compiler time warning, the Java compiler even allows the creation of classes that implement an annotation.

3.2.3 Dynamic Proxies

The inbuilt dynamic proxy mechanism in Java was available since JRE version 1.3 [19]. It provides an automatic way for implementing the Proxy Object pattern to handle method calls to a given object with an `java.lang.reflect.InvocationHandler` object while the object is still accessible as it was an object of given interfaces. However, the mechanism is limited to objects of classes that implement interfaces only, and the generated proxy object will no longer be an instance of the original class, and thus can neither be referred through this type.

Many DTO and entity classes are actual classes that do not implement an interface with all the setters and getters they possess. This is why, in most cases, the original proxy mechanism in Java would not be a solution for proxy objects. Instead, a third-party library with dynamic code generation capabilities can be utilized in order to generate proxy objects for any given non final classes. The dynamic code generation libraries are focused more closely in Section 3.5. These are usually also referred as dynamic proxies. The technique is well known and generated dynamic proxies also work as of the mechanisms that JPA implementations use for lazy loading of the data in entity object when properties other than the primary key are first accessed [20]. Spring also uses these class proxies by default for the implementation of its IoC mechanism to proxy beans for which interfaces injection is not used.

Java 8 will provide a language concept for method reference but this, however, is not a runtime concept. Instead, the implementation of method references as well as the lambda expressions in general are both just a shorter syntax for an anonymous class implementing a single method. Use of lambdas will be limited for single method interface types only. Hence, even though tempting and convenient to the outside,

the method references can not be used in runtime mapping of getters and setters. [21]

Mapping with proxy objects can be made null-safe in a way that `dto.setStreeName( entity.getCustomer() .getContactAddress() .getStreeName() )` would never cause `NullPointerException`s since it would only define the property path for resolving the mapping and each call would be handled and each return value would actually be created with an `InvocationHandler`. In the case before the `entity` and `dto` would be proxy objects and a call to `getCustomer()` would actually be handled in an `InvocationHandler` that would return a new proxy object with knowledge of the referenced path. Later on, the call to `getContactAddress()` would also be handled similarly, returning a new proxy object, that in turn, would contain a property path reference from the origin. These dynamic proxies can implement an additional interface to which they can be casted in the invocation handler of the setter method to access the mapped property path.

Even the dynamically code-generated proxies can not, however, extend final classes. This applies to, for example, all the basic wrapper types in Java, `java.lang.String`s, enumerations and primitives. It goes without saying that this concerns most of the types traditionally used in DTO classes' fields. In order to pass the information to the invocation handler of the setter method with these types, there is, however, one option. For these final types, one can use statically, thread-locally stored enumeration values as placeholders for the actual proxy objects. This would mean that, for example, the first getter method returning an `int` would return 0 and the second 1 and so on. The setter invocation handler may then access this storage to access the actual proxy. This kind of enumeration can be used to all basic Java types but it should be noted that some of them have very limited range of distinct values: for example boolean type has two and enumerations theoretically only one. Given the thread-local storage, these proxy references can not be mixed with similar mappings performed in parallel, but because of limited enumeration values, these proxy placeholders may not be stored in variables. This idea is based and an implementation for such enumeration value based proxies can be found for example in Lambdaj project [22].

As a limitation of the type safety provided by proxy mapping, in a situation where inner conversion between non assignable types needs to be performed, dynamic proxies can not be used without additional helper APIs. One possibility is to introduce a simple static generic helper method that converts any type to required type by generating a new proxy of that type with the property path from the original proxy. Then the need for conversion would be determined by noted non-assignability in the invocation handler of the setter method. Another considerable limitation is

that with proxy objects, only properties available through getter and setter methods could be mapped as it is not possible to proxy access to public fields.

3.2.4 Domain-specific languages

DSLs are used for their better expressiveness with more appropriate domain-specific notation or possibly for their better error handling, analysis, verification, optimization, parallelization or transformation capabilities compared to traditional General Purpose Languages (GPL), such as Java [23]. DSLs can be used in vertical manner to create whole applications for a specific domain but also in a horizontal manner to access a certain technical domain, such as specifying behavioral test cases, building database queries or, in our case, mapping DTO and entity classes.

Java programming language lacks the concept of a property, a combination of an attribute, setter and getter, which is a core concept in DTO and entity mapping. Because of this, properties can only be referred with `java.lang.String`s, which leaves room for errors since the referred property may be renamed or removed. Secondly, as stated earlier, there neither exists a way to safely refer to a method in a way that this reference could be used in meta-programming to refer to a getter or setter.

Thirdly, there exists no expressive way to refer to a chain of getters or a property path in a null-safe manner in Java programming language without the need to include conditional structures and either repetitive references or introduction of new variables. There was a proposal of adding a new `?.` null-safe reference operator to Java version 7 but it was rejected [24]. This either leads to a lack of expressiveness or to a possiblity of `NullPointerException`s. If, for example, one would like to map `customer.contactAddress.streetName` property path from `entity` object to `streetName` property of `dto` object with the possibility in mind that any part of the reference path might be null, the GPL Java code would look like the one in Listing 3.1.

Listing 3.1 A null-safe reference to a property path in Java programming language.

```
Customer customer = entity.getCustomer();
if( customer != null ) {
   ContactAddress contactAddress = customer.getContactAddress();
   if( contactAddress != null ) {
      dto.setStreetName( contactAddress.getStreetName() );
   }
}
```

However, a DSL with concepts for properties and null-safe references, could be

specified in a way that the syntax for the similar mapping would be just for example `dto.streetName = entity.customer.contactAddress.streetName`, making it a lot more expressive than the corresponding GPL expression. Some IDEs, such as Eclipse, can be extended to support custom DSLs so that features such as syntax highlight, autocomplete and automatic verification of property existence could also be achieved [25].

Instead of a fully custom made DSL, an easier option could be to extend some expressive dynamic languages directly. There also exists many such modern languages, such as Groovy [26], Scala [27] or Clojure [28], which are compiled in the very same Java intermediate byte language and run under the same JVM than the Java application. This gives them the advantage to both use and access Java code and utilize reflection for resolving type information.

3.2.5 Application Programming Interfaces

Mapping with Java APIs could be possible by using string references to properties and property paths. Such an API could also be extended with proxy objects to provide a more type and modification safe approach with the help of the autocomplete features of the IDE. The API could also provide, for example, callbacks for fully customizable logic where all the features of Java programming language could be utilized.

This kind of Java mapping API could also be accessible to other JVM languages but might lack some of their expressiveness, for example lambdas in Scala can not be used directly to replace callback interfaces in traditional Java APIs. Sometimes domain-specific Java APIs may also be referred to as DSLs.

3.3 Reflection

In this section we focus on the reflection API of the Java programming language [29] which can be used to access Java language structures and annotation specified meta-data in runtime. In addition to resolution of mappings, the Reflection API may also be used for accessing the values in fields or through getter and setter methods. These are covered in Subsection 3.3.1. Some limitations apply to the use of generics which are covered in Subsection 3.3.2. The performance of the reflection might also be a concern and is covered in Subsection 3.3.3.

3.3.1 Fields and Methods

The reflection API [29] provides name and type information for all the fields and methods, including parameter types, in a Java class. However, for constructors and

methods, parameter name information is not available. The reflection API also allows invocation of methods and getting or setting field values. [30]

The reflection information is available from all visibility levels but for invocations and setting or getting a field value, the visibility levels are obeyed by default. It is, however, possible to circumvent the visibility restrictions by explicitly calling `java.lang.reflect.AccessibleObject.setAccessible(true)` implemented in both `java.lang.Method` and `java.lang.Field`, and thus forcefully break the abstraction of an object. [30]

3.3.2 Generics

While the reflection API in Java provides the type information for arrays directly, generic collections are a more complex case. Generics were introduced in Java version 1.5 [11]. Basically, they function as nothing but extra compile time type safety checks and extended class file metadata, since in byte-code level all actions are still preformed on raw types because of a process that is known as type erasure [31]. On the other hand, this ensures that there is no runtime overhead and provides compatibility with earlier Java versions, making it possible to utilize Java 1.5 [11] or newer software libraries with generics even in source code targeted to Java 1.4 [32] or earlier versions. But it also means that at runtime it is not possible to determine the actual type parameter values used for an object of class having generic type parameters or within a method with a generic signature. This means that if, for instance, a generic purpose component with a `java.uitl.Collection<?>` interface parameter is handling instances of for example `java.util.Collection<Customer>`at runtime, the `Class<Customer>` instance is not accessible and must be determined explicitly. [33]

Having said that, however, the generic types and their bounds specified in classes, methods and fields are accessible, meaning all the information available in class and interface definitions at compile time, including method return types and parameters types, can be determined by reflection [33]. This means that if, for instance, a class `Customer` has a method with signature `public java.util.Collection<Order> getOrders()`, the `Class<Order>` instance can be accessed using reflection. It is still possible, however, not to use generics or only to specify bounds for the generic type parameters. For example, the method signature could be `public java.util.Collection<? extends OrderInterface> getOrders()` where the actual type parameter could be any type extending or implementing `OrderInterface`. Or the code could simply be written in Java 1.4 [32] style omitting the type parameters as `public java.util.Collection getOrders()` where reflection would only be able to suggest `Object` as the generic type.

When reflecting through abstract classes or interfaces that have unspecified generic type parameters, the actual parameter types are also not accessible at runtime but reflection does provide the same amount of information which is available to the programmer at coding time, including the bounds of those types. If we had, for instance, a generic `Refund` interface that might have different `RefundTarget` interface implementing targets, such as classes `Order` or `Subscription`, with signature `interface Refund<Target extends RefundTarget>` and that had a method `public Target getTarget()`, the reflection could only access the `RefundTarget` interface if we reflected the method through the interface type directly. Only when reflecting the method through `class OrderRefund implments Refund<Order>` class where the generic type is specified, the `Order` class would become accessible, which is exactly what a programmer would see as well.

3.3.3 Performance

Traditionally, in Java the method calls or field value access through reflection are generally considered tens to hundreds times slower than direct call to methods or fields [30]. This is caused by the additional steps the Java Virtual Machine (JVM) needs to take including security manager calls and virtualization checks which it could omit in verified byte-code. The performance has been stated to increase slightly over the development of JVM from version 1.3.1 [34] to version 1.4 [32] [30]

In a modern HotSpot JVM architecture, however, in what is called Just-In-Time (JIT) compilation, the code is initially interpreted and selectively optimized based on runtime profiling analysis [35]. With JIT it is actually difficult to reliably test the performance of method calls, since among a lot of other optimizations, most simple method calls are actually eventually inlined into byte code on the side of the caller. What is more, the JIT compiler also automatically turns `java.lang.reflect.Method`s often invoked into proxies that actually call dynamically compiled byte code, or it may be able to inline them as well, given that this call repeats a certain number of times. The server version of the JIT compiler provided by Oracle, used with most Java EE applications, is targeted to speed up usually long running application with aggressive optimizations and needs up to 10 thousand profiled invocations prior this optimization in order to correctly analyze the normal use of the application after the start-up and warm-up periods, whereas the client version is targeted to fast start-up and will by default do the optimizations based on just 1 500 calls observed through runtime profiling. Optionally, in so called tiered mode, the JVM uses both of these compilers, first in client mode and later on in the server mode. [35]

Because of the inline optimization, a correct performance measurement should access the internal state of the object, which is actually the case for all setter and getter methods used with DTO and entity classes. Also, it should be noted that creating objects and especially the caused garbage collection is a costly operation and may affected the measurements, so creating new object instances should be avoided during the test. In addition, the lookup for reflection elements should only happen once, and because of the optimizations done by the JIT compiler, invocation thresholds and lazy class loading used in Java on the other, there should be warm-up phase with at least 10 000 calls that are not counted to the test results. Following these guidelines with JVM version 1.7 in server mode, test results done for this thesis with one million method calls show on average roughly a 660 times performance difference (0.0180ns vs. 11.8ns per call) between direct method calls and reflection calls with simple methods that the JIT compiler may inline, but only about a 6.4 times difference (1.95ns vs. 12.5ns per call) with methods that access the internal state of an object. These measurements are supported by the measurements done by Dimitry Buzdin where the mean time in server mode for direct method call is measured to be around 3.92 ns and 24.0 ns with reflection, making roughly the same 6.1 times difference [36].

With the use of JVM options `-server-XX:+PrintCompilation -XX:+UnlockDiagnosticVMOptions -XX:+PrintInlining` the compilation and inlining results of the JIT compiler may be analyzed [36]. For example, the final compilation in the warm-up period for the test code is presented in Listing 3.2. What is notable, is that most of the reflection related steps are inlined or intrinsic, meaning they are replaced by native code supporting the best guess by profiling [36], but access checks are considered too big for inlining. However, using the `java.lang.reflect.Method.setAccessible(true)` will not reduce the runtime, and will, in fact, result in a similar JIT compilation. Even if we would further fine tune the execution by setting `-XX:MaxInlineSize=100`, the first too big call would eventually resolve to not being executed at all and the second would be executed fewer number of times than the minimum optimization threshold, even with it set to a low value such as `-XX:MinInliningThreshold=100`.

All this indicates that the reflection call may not become any more optimized. On the other hand, JIT optimization does reduce the reflection invocation runtime significantly, since if we would prevent the JIT from inlining the invoke-call by setting `-XX:CompileCommand="exclude java.lang.reflect.Method::invoke"`, the invocation would take 76.82 ns which is about 6 times more than with the JIT optimization.

The measured, relatively low, about 6 times difference in optimized reflection

Listing 3.2 The JIT compiler output for a reflection method call

```
 TestReflectionPerformance::callReflectArgs (52 bytes)
@ 10   java.lang.Integer::<init> (10 bytes)   inline (hot)
@ 1   java.lang.Number::<init> (5 bytes)   inline (hot)
@ 1   java.lang.Object::<init> (1 bytes)   inline (hot)
@ 29   java.lang.reflect.Method::invoke (63 bytes)   inline (hot)
@ 15   sun.reflect.Reflection::quickCheckMemberAccess (10 bytes)
inline (hot)
@ 1   sun.reflect.Reflection::getClassAccessFlags (0 bytes)
(intrinsic)
@ 6   java.lang.reflect.Modifier::isPublic (12 bytes)   inline (hot)
@ 22   sun.reflect.Reflection::getCallerClass (0 bytes)   (intrinsic)
@ 37   java.lang.reflect.AccessibleObject::checkAccess (96 bytes)
too big
@ 50   java.lang.reflect.Method::acquireMethodAccessor (44 bytes)
too big
@ 57   sun.reflect.DelegatingMethodAccessorImpl::invoke (10 bytes)
inline (hot)
@ 48   java.lang.Integer::intValue (5 bytes)   inline (hot)
```

based access compared to direct Java source becomes, of course, amplified by the additional code and method calls needed for DTO and entity mapping compared to a straightforward Java code solution. But since the implementations of getters and setters are basically really simple, their cost is practically close to zero, and the difference is linear, as high optimization as the 6 times difference is not to be expected in real applications, but rather should be considered as the maximum performance against pure Java code where the non inline-optimized 36 times difference could be closer to the truth. This is because the JIT compiler balances the use of resources with the running application and only focuses on the real, usually rare hot spots of the application [36]. Although, probably, even a 100 times difference would hardly be relevant or even noticeable in the total performance of a Java EE web application where the largest portion of the runtime is generally consumed waiting for database queries, other external services or rendering a view, the expectantly low priority for optimization by JIT compiler actually makes performance tuning a more important factor in a general purpose mapping component.

3.4 Caches

The Java programming language itself lacks the concept of a property. It is merely based on Java Standards that a Java class should have getter and setter methods named according to the attribute they provide access to. For example, the JPA

standard [5] relies on this standard and involves reflection based property mapping. The lack of language support basically means that there is no direct way of mapping the getters and setters to property name or vise versa through reflection. To get this information, one must therefore go through all the public methods of a class and interpret their signature against the Bean property standard. It goes without saying, that this kind of reflection is costly, should only happen once, and therefore involves the use of caches.

Changes in class interfaces, interfaces, annotations, method signatures or fields are not supported by the HotSpot feature in the JVM. Only the implementation of already loaded class may be changed at runtime [37]. These changes do not affect the type hierarchies, fields, getters, setters, annotations or other important factors when resolving mappings between entity and DTO classes. Furthermore, if we also consider other forms of configuration static, it is a safe assumption, that the mappings remain unchanged throughout the whole runtime of the application in a production environment. Based on this assumption, the performance of the mapping should always be linear.

While this assumption will allays be justifiable in the production environment, there are some exceptions, such as Dynamic Code Evolution VM related to the HotSwap project of Oracle which extends the HotSpot technology used in the JVM to allow dynamic code evolution in the means of adding and removing methods, fields and supertypes during runtime [38]. There is also a commercial JRebel tool [39], which can reload Java classes with class, interface or annotation changes occurring even during runtime without using throwaway class loaders [40]. Rather than that, JRebel integrates with the JVM as a plugin in order to avoid the delays caused by application re-deployments during development time [39]. In order to avoid costly repetitive reflections, many frameworks, including Spring and Hibernate, however, use just the kind of class caches that would become invalid after the Java types. For this purpose, JRebel uses byte-code level interception, cache invalidation and other customized means for interacting with the most common frameworks. For smaller, non-supported frameworks, JRebel provides a plugin mechanism, that can be used to invalidate the cache entirely or for particular class [41]. Therefore, given that the mapping component can provide a way to invalidate its class caches and other depended caches, the assumption of type immutability can safely be made.

3.4.1 Early-Work and Lazy Initialization pattern

Because the results of resolution and possible code generation can be cached, depending on how the mapping is done, the early work could be used in the resolution phase and possible dynamic code generation. For compile-time components, this is

the only possible way to perform the resolution and code generation. For runtime components, this would shift the delay caused by resolution and possible code generation from the actual use of a mapping component to its first initialization, which could be done, for example, during the start-up of an application. For a large application, this could possibly add some notable waiting time in the compilation or start-up of the application, but on the other hand, by resolving all the mappings at once, it would ease the testing of correctness of the mappings.

From the mapping configuration, early work requires the knowledge of all the class pairs, which could possibly be mapped against each other during runtime. This can be achieved by all the mapping techniques discussed earlier, although when doing multiple mappings for the same type, mapping with annotations would turn more complex by the need of wrapper annotations. As a tradeoff, for the most simplest scenarios, mapping could be done automatically based on field names and types of the two Java types and for these cases, introducing a need to explicitly define all the class pairs in a mapping configuration could add an extra overhead to the coding phase.

Alternatively, Lazy Initialization pattern, or Lazy Load pattern [12], may also be used to resolve the mappings when they are first needed between two given Java types. This option, of course, remains only for runtime mapping components. Lazy execution adds some overhead to the first conversion, although the difference should remain at most in milliseconds and would hardly be notable by the user. As well as, with early work, the results would be cached so that the second time, a conversion between the two types is needed, there would be minimal overhead. By shifting the resolving phase up to first use, the Lazy Execution Pattern approach also shifts the point of noticing an error in the mapping configuration, and thereby introduces the need for comprehensive testing. These factors in mind, the choose between these two patterns is primarily a tradeoff between the amount of needed mapping configuration and its testability.

3.4.2 Thread-safety with Singleton Caches

In a Java EE component container, the singleton scoped beans are shared between all requests, meaning a runtime mapping component is likely to be used simultaneously. All caches used by the component should therefore be thread-safe. The easiest possibility would be to utilize the inbuilt mechanism in Java, attribute or class synchronization, but given that DTO and Entity mappings are to be used in most service level methods, this methodology might reduce the overall scalability of the application.

To avoid unnecessary synchronization in the Java programming language, two

alternatives exits: static initialization and the double-checking idiom. The static initialization blocks are called in thread-safe manner when the JVM first loads the class. They can also be used in a lazy manner by introducing a field that references another class containing the static initialization code which, because of the lazy class loading used in Java, gets executed only when first needed. Static initialization based class caches also work with HotSwap extending technologies that are based on throwaway class loaders, since the static blocks are called, and cache hence refreshed, every time a new class loader accesses the class.

Static initialization works for class cache and other collection initialization in a static manner. However, for component configuration based mappings which may vary based on object instance, static scope is limited. For object based caching purposes, double checking idiom [42] can be used. It, however, requires the use volatile keyword with the initialized field so that all threads can see the changes, which in turn sets the minimum requirement for Java version to 1.5 [11].

3.5 Dynamic Code Generation

When it comes to the actual access of properties, through their setter and getter methods, in the inevitable phase of copying of values during runtime, as discussed earlier, significant performance losses are to be expected when using reflection based method invocation, and the JIT compiler is most probably unable to optimize those operations. One way to overcome this performance issue, which is also used by the most component implementations, is to dynamically generate and compile the code that does the actual copying of values in runtime. By doing so, not only does the access to properties get optimized but the logic in between these operations gets minimized as the generated code can more or less represent the result of a straightforward manually written code used to map DTOs and entities.

For this task, there exists a variety of libraries, which differ mainly in the features they offer and their APIs. Among these libraries the performance of the compiled code, however, does not have a significant difference other than compared to use of pure reflection [43] [44]. These tools include byte code provider libraries such as Cglib [45] which has many byte code manipulation related projects depending on it, such as Byte Code Engineering Library (BCEL) in Apache Common [46], Javassist [47], ASM [48] and the standard `java.tools.JavaCompiler` [19] which has been available in JDK since Java version 1.6 [49] through `tools.jar` but is not bundled with JRE by default.

Some of the libraries, such as ASM, are very low level and possibly more efficient but their use requires the knowledge of Java byte code structures. On the other hand, for example, Javassist can compile method bodies directly from Java source

strings presented in source level 1.4 [32] and the standard Java compiler will be able to compile whole Java source files. In addition to compiling new classes, many of the byte code providers are also able to instrument existing code. Because the byte code providers depend on the Java byte code format, they also depend on the `.class` file version and need to be updated as new Java versions are published with new byte code format. For example, Spring Foundation needed to patch ASM with Java 8 support to meet their delivery schedule for Spring Framework version 4 [50] and Hibernate needed to change from Cglib to Javassist in 2010 because Cglib development had slowed down [51].

There are some challenges when using dynamic code generation. Firstly, debugging the generated code might be really difficult unless the code can not be represented in the original Java source code format and thus run and analyzed separately. Other related issue is that the compiling errors these libraries produce, do not necessarily specify the section of the code that produced the error. Exceptions, such as `NullPointerException`s thrown from within generated code are also almost impossible to trace because the stack trace will not have line numbers. The fact that most byte code providers operate in early Java source code levels, usually 1.4, also means that there is no direct support for generics and special attention must be paid to explicit transformation between primitive and wrapper types. Therefore, caution and practices that prevent such errors prior the actual dynamic compiling phase, should be applied when writing code generating code.

A possible solution for overcoming the problems related to the use of byte code providers is to introduce an intermediate, type and null-safe API for building the code. Such an API would throw exceptions for the most typical errors before the code gets compiled. This also serves as an layer of abstraction making it easier to change the underlying byte code provider.

4. REQUIREMENTS

In this chapter the requirements for a generic DTO and entity mapping component are specified based on the environment and use case analysis in Chapter 2. The requirements are listed based on their urgency and occurrence in a software project utilizing the component. The functional requirements for the component are listed and their importance is weighted in Appendix A. In this chapter the requirements are referred to with superscript numbers in round brackets.

In Sections 4.1, 4.2 and 4.3 the requirements related to the used frameworks and techniques are discussed. After that, in Section 4.4 the requirements relating to ease of use of the mapping configuration are discussed. In Section 4.5 the requirements relating to different features in the mapping component are presented. Finally, in Section 4.6 the requirements related to customizability are explained.

4.1 Maven Support

Maven is an industry standard build automation tool used by Dicode in most of its Java EE projects. Among its other features Maven provides dependency management for software packages. Single packages in Maven are called artifacts. Artifacts may depend on other artifacts and it is the responsibility of Maven to automatically download the required versions of these artifacts from different repositories upon the build of the software based on the configuration file of the Maven project. This provides flexibility in the build automation process and software component deployment since all dependencies are defined and managed in a single XML file rather than maintained as vast set of jar files with the project.

In order to provide this flexibility the component should be available as a Maven artifact $^{(1)}$. This also serves as an easier method of customizing or branching the component itself while the customized or fixed version could then easily be deployed in an internal repository and taken in use simply by changing the version of the component in a Maven configuration file. Since Dicode uses an internal Maven repository to ensure the availability of certain Maven artifacts, the availability of the mapper artifact in the central Maven repositories $^{(2)}$ is, however, not considered as a strict requirement.

4.2 Spring Support

The component must not have direct dependencies to any particular web framework or EJB container, since this would largely limit the generality of the solution. Any artifact dependencies to these frameworks should only be delivered in separate artifacts or be optional with the ability to compile without these dependencies available. For Dicode, Spring is the most commonly used web application framework for Java EE application. Thus, the solution must be usable at least in a Spring environment (3) but should be customizable enough to be used with any other framework as well (4).

It can be safely assumed that mappings between existing DTO and entity classes remain static throughout the whole runtime life cycle of the application. Thus, the usage of application singleton scope would be the most preferable because of its subtle use of memory resources and the ability to make single time lazy mappings between these classes. The inversion of control (IoC) mechanism used in Spring also makes it very easy to change this design decision afterwards if necessary [1]. Notably, in application scope the component will be used concurrently by multiple users and must thus be thread-safe (5). However, this should not involve unnecessary use of synchronization which could have a significant effect on the scalability of the application [52].

Furthermore, the mapper component should be available as several customized components for different purposes. For example, if the service level of the application was divided in different packages based on entity aggregation groups of the domain model, the most likely use scenario for the component would be to have a separate, possibly customized, implementation of the mapper for each aggregation group. This would be done in order to achieve better segmentation of responsibility as well as minimizing the inter-dependencies between these packages. Since interface injection is one of the most often used forms of autowiring required components in the IoC mechanism of Srping for non XML based configurations, as a requirement the component should implement an easily extendable interface (6), the variations of which could then be injected as different customized components all still providing the basic functionality.

4.3 JPA and Hibernate Support

Since mapping entities and DTO classes is not essentially different from mapping any two Java classes, there are not many requirements that relate to JPA nor the persistence implementation directly. However, when converting DTOs to entities, there is a need to automatically fetch entities by their primary keys (7) which would

otherwise involve a great deal of manual work. This involves determining which class represents an entity and which property of it represents its primary key. The mapper component should also separate cases where multiple entities are fetched by array of collection of primary keys (8) in order to minimize the number of queries.

JPA itself specifies a common interface for fetching an entities by primary key in `javax.persistence.EntityManager`-interface the current, e.g. thread-local, version of which is provided by an implementation of `javax.persistence.EntityManagerFactory`-interface. Through this API the primary key meta-data information is also available [5]. These interfaces work as delegates for JPA implementation specific interfaces such as the `EntityManager` corresponding `org.hibernate.Session` obtained by `org.hibernate.SessionFactory` in Hibernate or `oracle.toplink.tools.sessionmanagement.SessionManager` in TopLink, for example [6][7].

For the purpose of generality, however, the mapper component could ideally define an interface for the features it needs, which are determining the primary key property and fetching entity by primary key or multiple entities by multiple primary keys, and then provide a JPA specific implementation of this feature as a separate, optional artifact (9). This way the implementation would not be directly JPA dependent, the persistence and ORM implementation could be changed to other than JPA or a possibly to a more efficient implementation dependent solution could be provided. Because Dicode commonly uses Hibernate implementation in its Java EE projects a readily existing Hibernate compliant adaptation (10) would be preferred but the implementation should support any other JPA implementations (11) as well.

4.4 Ease of Use

The fundamental goal for the existence of the mapper component is to reduce the amount of work a developer has to do when converting between DTOs and entities, and by so doing avoiding boilerplate code and aligning architecture. In order to be used and accomplish this goal, the mapper component must be easy and practical to use minimizing the amount of code and configuration. Whenever possible the need for configuration should be circumvented by providing defaults that work for most scenarios and that can be overridden. This applies both to taking use of the component and to using it in source code level.

4.4.1 Annotation-driven Configuration

As described in Section 3.1, there are multiple options for defining the mapping configuration. Referring to practical experiences at Dicode over the past few years, whenever there has been a chance to choose, we have found annotation-driven

configuration much more convenient compared to XML-driven ones [12]. This has been shown with for example Spring and JPA where in previous versions for example the bean definitions, transactional proxies and entity mappings needed to be defined in separate XML files. Annotation-driven options have been used since they become available. The inconvenience discovered was mostly caused by the separation of the configuration from the related code, the configuration that, in many cases, is not actual configuration at all, because there most likely exists only one way of doing it.

In practice, the XML configuration that could be replaced by annotation driven one, meant that whenever there was a change in the code, for example a new transactional service method or an injected bean dependency was added, the related XML files needed to be searched for and changed. Especially when there were multiple changes in code requiring changes in XML definitions, some of them were easily forgotten resulting in run-time exceptions continued by a possible rerun with new discoveries. This process was impractical, time consuming and therefore also easily resulted in architectural anti-patterns, such as beans with multiple and even non-related responsibilities simply due to the fact that extracting the added functionality to a new, separate bean would have required defining it, the related transactional proxy and dependencies in these XML configurations.

When it comes to domain-specific languages or source code mapping, many of the features are shared with XML. They are as separate to the mapped source code as are XML mappings and would cause additional maintenance burden. Additionally, domain specific languages would add a learning curve by introducing a new language the developer should learn in order to do a simple task. This could easily lead to situation where only a handful of developers would utilize the mapping component and lead to the same architectural concerns and before.

In an annotation-driven approach annotations get inserted as the related code is written making it easier to remember. They are also much more likely the be updated when changes in the related code occur as they exist right beside the changed source code. Additionally, if the configuration causes changes in the behavior of the application, which for example transactional proxies do, their presence as annotations in explicit and visible form is much more likely to support sense-making for a developer maintaining the project.

XML mapping [15], mapping by API [16] or DSL mappings [17] are preferable where multiple configuration could co-exist. However, with DTO and entity mappings there are only one way of mapping each particular DTO class with an entity class. XML configuration could be argued over the two other since it could provide better tooling support. On the other hand mapping with proxy objects [18] would make the mappings change as code gets refactored which could slightly reduce maintenance

and testing efforts. DSL or other scripting expressions could allow specification of conditional mappings which may depend on the state of the mapped object (19) but on the other hand this could also be achieved by a custom converter and these kind of cases are rare.

One arguable impediment for the usage of annotations in DTO mappings could be the additional created dependency to the library containing these annotations in the case that the same DTO classes are used among multiple distributed systems via serialized form such as with the Remote Method Invocation (RMI) technology. The use of RMI in Java EE applications is, however, usually little, and dependencies could be minimized by defining them in a separate artifact. Where this could be an issue, XML configuration could be used instead.

4.4.2 Convenience in Mapping

When looking for a convenient way of doing DTO and entity mapping, we could look for solutions used in JPA for a similar but ORM-related purpose. For example, the mapping in JPA shows a good example of providing functional and overridable defaults: Given that the property name and type of an entity matches those in the database scheme, no configuration is needed. However, if the property name differed or a more complex data type should be converted, this default behavior can be overridden by annotations.

This kind of mapping of matching name and type would be the most likely case in entity and DTO mapping as well and automatic mapping (13) could be similarly expected by a developer used to working with JPA mappings. In practice this would mean that adding a new property to an entity and those DTO classes in need for it would be enough for making the mapping meaning basically zero configuration and less forgotten mappings as new properties are added.

Especially with aggregate structures the camel-case property naming convention widely used in Java could also be useful in discovering mappings (14). This would mean that e.g. `customerName` would be mapped with a property path `customer.name` and similarly `customerContactAddressStreetName` could be mapped directly with `customerc. contactAddress.streetName`. With a simple rule such as the shortest property name wins in path decisions, ambiguous mappings such as `customerProfileId` mapping with either `customer.profile.id` or `customerProfile.id` with the same type could be solved. Using camel-case mapping as the default behavior, these kind of conflicts could always be solved by overriding the mapping.

4.5 Feature Requirements

Feature requirements were discovered based on the use case analysis in Chapter 2. In this section these requirements are listed in the order of their importance.

4.5.1 Bi-directional Mapping

In editing use scenario the same DTO class is mapped with an entity in two directions. In these cases both ends of the mapping posses necessary getters and setters for bi-directional conversion and with the JPA support discussed earlier, related entities could be fetched by their primary keys automatically. In order to minimize the repetitive work and errors caused by it, mappings should be done only once for each DTO and entity pair (20).

At the same time mappings must be customizable in such a way that mapping to one way or the other or both can be prevented (21). This is to allow manual fetching of related data and checks when some parts of the entity model must not be edited freely. Also properties having only a visible getter or setter should automatically make the mapping the suitable way only (22).

While annotations, as discussed earlier, are the preferable means of configuring the mappings, there may also exists situations where DTO classes are defined in external jar files and can thus not be edited with these mapping annotations. This kind of need would occur when e.g. Service level provides presentation DTOs, such as `javax.faces.SelectItem`s directly. With the use of two-way mapping and mapping annotations independent of the end they are defined in, these kind of limitations could be overcome by defining the mappings in entities (23).

4.5.2 Aggregation Mapping

Multiple use scenarios for DTOs, such as viewing, creating and editing, include flattering or altering the aggregation structure of an entity. This infers that rather being single property pairs, mappings need to be property path pairs where e.g. property named `customerName` could be mapped with `customer.name` resulting, simply put, in `getCustomer().getName()` and `getCustomer().setName(...` equivalent calls (24).

The `getCustomer()` example above, however, is not quite valid. The mapping component can not expect the containers to be non null and should never throw `NullPointerException`s in the case where the customer-property of the source object was null (25). Actual causes for these kind of runtime exceptions in a generic purpose component could be very hard if not impossible to trace. Instead, null values should be handled in such a way that any null value within the conversion

path should result in a null value in the end result without exceptions thrown when reading the value.

Setting a value in a null container creates another story. When a new aggregation group is created, related entities default to nulls and should be either fetched by their primary keys (26) or created with default constructor (27) so that they would be saved either manually or by cascades set to the JPA mappings. When editing an aggregation group with a DTO having a property holding the primary key for the container object, the appropriate option would be to load the entity for editing and then update the fields mapped to its fields.

It is also possible that the structure of aggregations is not flattered but rather repeated in DTO class. In such cases inner conversion (28) between the DTO-property and related entity should be applied where requested. When building such DTOs from entities, the null container problem is reversed: Now the contained DTO should be created with its default constructor and properties set to match the related entity. More likely than the DTO itself, its properties may be DTO classes or value objects specified in external packages that can not be edited. Thus, when using annotations, there should be a possibility to map these inner conversions from the container (29).

4.5.3 Type Support

The component should allow the use of coding practices that prefer the use of abstract types over concrete ones, for example the Dependency Inversion Principle of SOLID Design Principles [53]. This requires the component to support interfaces and abstract types in both DTO and entity classes so that the implementation type can be specified.

The implementation type could be defined directly case by case with a concrete Java type on mapped class pair basis (30) or indirectly with an type alias (31). Especially with annotations, type alias and specific mapping of aliases to concrete Java types would be preferable, since it would maintain this abstraction and especially compared to defining a reference to a Java Class, removes the dependency and makes the object more easily transferable over system boundaries. Additionally, it would be convenient to be able to define general default implementations for given abstract types (23) so that the amount of work put in these definitions would be minimized in the quite typical case where there exists only one implementation for given interface.

While the ability to define the implementation type also works as a solution for generic classes with open type parameters that are used in properties, it would be preferable if the typing system used in the component would maintain as much

information on the actual generic type parameters as possible. This information is not accessible in `java.lang.Class` objects because of the type erasure, as discussed earlier in Subsection 3.3.2, and therefore introduces a need for an abstract generic type implementation in the component. When anonymous implementations of such generic class are used in mappings, the generic type information is available and enables automatic determination of actual types of properties with an open generic type (33).

4.5.4 Type Conversions

Data type decisions are usually consistent throughout an architecture tier but sometimes there are differences between the tiers. Most obvious ones are the ones related to the use of primitive or boxed types such as `int` vs. `Integer`. For example the entity model might use boxed types because the values might be null at certain point prior saving the entities but if not null constraints are in place, the DTOs meant for viewing might hold primitive types. Similar differences might also exists with more complex types, for example, through the use of an external date library, such as Joda with `org.joda.time.DateTime`, `org.joda.time.LocalDate` types versus `java.util.Date` equivalent in Java, or the use of `java.lang.Long` vs. `java.math.BigInteger`. In some cases String values might be used instead of enumeration values in entities or vice versa. Some external libraries such as JAXB also require using their own types.

The actual type conversions over these kind of inconsistencies are usually handled the same way between two given types throughout the application and are likely to add some overhead if done manually. Thus, these conversions can be considered as general concerns that can be centralized by automatizing them in a generic DTO and entity mapping component. With the ability to define new type conversions (34) this kind of feature also enables the user to easily extend the mapper component with additional features such as automatic localization when converted between a database internationalization entity or enumeration and a String or even automatically saving entity related localization values when converting with DTO holding the localizations.

The component should provide basic conversions such as conversions between boxed and primitive types automatically (35), but should allow user-specified conversions as well as replacement (36) of the readily defined ones. As with aggregations the conversions should be null-safe so that null values should remain null and ignored when converted to primitive properties (37). In order to avoid accidental mapping errors the type conversions should only be applied when specifically asked to. If the component would require explicit case-by-case definition

of which type conversion to use with each property, which could sometimes be necessary (38), the benefits of the centralized handling of this concern would be at least partially lost and the manual overhead would remain considerable. The component should therefore preferably allow defining general type conversion to use automatically (39).

4.5.5 Collection and Array Support

Especially in creation, viewing and editing targeted DTOs, aggregation structures often also contain collections of related data. Inner conversion should therefore be applied also to related collections (40). In DTOs these are often represented by ordered collections, such as various implementations of `java.util.List` or primitive arrays. In entities, however, the data is often represented by non ordered `java.util.Set`s. The generic converter should be able to automatically convert content between different implementations of `java.util.Collection`s and primitive arrays alike (41) with inner conversions applied. When using primitive arrays or `java.util.Colelction`s in a type-safe manner with generics the contained type information can be automatically determined without explicit specification (42).

For the entity to DTO mapping direction the component should provide a way to specify the ordering of the data (43) as if a `java.util.Comparator` was applied to the collection of DTOs. This is needed in editing and viewing scenarios to show the related data in a fixed order. To further automate this process, annotations could be used to specify which properties of the DTO should be used as basis for the comparison in certain relative order to each other and in either ascending or descending order (44). This would be possible given that those properties implement `java.lang.Comparable`, which most of the simple data types used in DTOs do. In the most simple cases the DTO collections could only hold one property of the corresponding entity, such as a String holding a name or an Integer representing a primary key. Since these are `java.lang.Comparable` as well, such values could be automatically compared to provide a fixed ordering. Such ordering should, however, only be automatically applied when converting from a non-ordered source.

When creating a new entity, DTO is converted to an entity and the order in which the data is added, is irrelevant. However, when editing an existing entity, the values in such collections or arrays should be synchronized by using the primary key of the entity and the corresponding property in the DTO class (45). Thus, when mapped, to help with the most frequent synchronization need, the component could assume that those related elements not found from the collection of the DTO should be removed from entity and those not existing in the collection of the entity should be added (46). It would be left to the cascade mappings of the JPA to decide whether

to actually persist the new elements when added or delete when removed from the collection. If the DTO would not contain the primary key property, when mapped, all such related data should first be removed and then added as new, which would in many cases lead to a similar result.

Instead of pure aggregations, collections and primitive arrays are also used to connect the DTO to related entities by just their primary keys, especially when it comes to many-to-many mapping. In these cases the collection or array is usually an aggregate of a single property in the related entity. This could be mapped similarly to paths in aggregation mapping. For example, a `Long[] customerIds` property in a DTO cold be mapped to the `Long id` property of the `Customer` accessed through `Set<Customer> customers` property of the related entity. This way, when converted from a DTO to an entity, the related `Customer`s would be automatically fetched by their primary keys in a single query.

For viewing purposes these collection contained path aggregation properties might also be other than primary key and mapping path could be longer than just two parts. This is where collection projection is needed (47). An imaginary `OrderViewDto` might, for example, contain a `java.util.List<String>` collection property containing the full names of the `ContactPerson` of a related `Vendor` for each `OrderItem`s, i.e. path `orderItems.vendor. contactPerson.fullName`. Such mappings could of course be applied in entity to DTO direction only.

Sometimes it would also be useful to be able to filter collection values (48). For example, a contained entity could have a boolean property telling whether it is actually deleted and should not be visible to the presentation layer. Such filtering would need to happen in the source end before the actual conversion and should be definable for both ends.

4.5.6 Field and Getter/Setter Support

Among stabilizability and the existence of a default constructor the Java Beans specification demands the definition of getters and setter for all properties [54]. However, some techniques, such as JAXB, also allow the use of public fields as in basic Java classes often referred to as Plain-Old-Java-Object (POJO) [15] [55]. Taken that DTOs might be converted to JAXB classes and JAXB classes might be treated as DTOs, the generic component should support POJOs meaning it should be able to access properties through visible `get`, `is` and `set` prefixed accessor methods as well as fields (49).

Although possible with reflection [30], the component should never attempt to break encapsulation by making forced calls to non visible methods or altering non visible fields (50). Because the accessor methods might alter or validate the data and

are higher in abstraction, the component should use getters and setters if available and only fall back to fields only if related getter/setter are not defined. If only a getter method for property is defined, the property should be treated as only readable, and accordingly only writable when only a setter method is available. Visible fields can be considered readable and writable. However, if the property is only mapped in one direction or not mapped, these rules must be respected above the visibility of properties.

Where annotations are used for configuring the mapping component, they should be equally applicable to either the getter method of a property or a field with the property name (51). This applies also to private fields. This behavior would be consistent to annotation configurations in JPA [5] and would thus be expected by developers used to working with JPA.

4.5.7 Immutable Object Support

As with getters and setters, the component can not expect all the mapped objects to have a default constructor. The enforced use of custom constructors is usually related to immutable objects where the state of the object is initially created with a constructor and all mutator methods return a new instance of the object. This pattern is often useful and requires the component to support custom constructors (52).

A class can have multiple constructors targeted for different kinds of use. It should be possible to define which constructor to use by defining the data types of the constructor arguments or marking the constructor to be used in DTO conversion by an annotation (53). The immutable object could also be a value object defined in external library that could not be altered. For this purpose the decision of the constructor should also be possible from within the container DTO (54).

As far as the mapping is considered, the parameters for the constructor are essentially similar to property mappings. All the features supported for properties should also be available for constructor arguments (55). Similarly to the decision of the constructor, also the parameter mappings should be possible outside the code of the immutable class (54).

4.5.8 Support for Graphs and Two-way Linking Structures

Data structures with two-way linking, such as two-way linked lists, hierarchical structures with parent reference, nets and so on, could easily cause infinite recursion or, in practice `StackOverflowException`s, if not handled correctly. This is especially notable with the aggregation related inner conversion support of the mapper component. Although two-way linking is rare in DTOs, it is still a possible

scenario and as with `NullPointerException`s the errors caused by it could be really hard to trace in a generic component.

To overcome this challenge, the component should hold a per conversion tree cache state of converted objects. A simple solution, that might not work with most complex data structures, such as nets, but would not require much state information, could be a special mapping to parent that would cause a reference upstream in the conversion call tree [56]. A preferable alternative solution that works for all kinds of structures would be to link all converted source references to target references [57]. For each new conversion occurring within this tree the mapper component could then check if the source has already been converted and use the readily converted result instead of new conversion.

4.6 Customizability

Part of the customizability required from the mapping component relates to every day use cases. Since DTO classes can be various, a fully generic solution might be an impossible goal. In addition to these customization needs the component should also be customizable for future unforeseen needs with interchangeable and extendable implementation.

4.6.1 Mapping Directions and Prohibiting Mapping

As discussed in Subsection 4.5.1 the mappings should allow two-way mapping from Entity to DTO and vise versa at once. However, in many cases some of the properties should not be mapped in this two-way manner. For example an `EmployeeOwnInformationEditDto` could hold information about the current monthly payment but should not allow updating that to the model. The same applies to some of the relations and their primary keys. On the other hand a `LoginUserEditDto` would probably contain a `password`-property that would be mapped to the setter-method of the entity hashing the password to be stored in the database. But when reading this information from the entity, the hash value should probably not be copied to the DTO and displayed in the view.

To also enable the two-way mapping, rather than specifying conversion direction related restrictions, these kind of restrictions should be specified by customized readability and writability of the property for the conversion purposes where the visibilities of the properties should be obeyed. Some properties in the entities or DTOs might also be preferred to be handled manually in the code even though they could be automatically mapped. For these purposes it should be possible to skip the conversion altogether [58]. As these cases are more of an exception than commonality, blacklisting the non-mapped properties still makes it a more practical

approach for a developer than requirement to explicitly map all properties. Hence newly added properties will also be automatically mapped.

4.6.2 Multiple Mappings

The reuse of DTOs for different purposes is usually an anti pattern. However, there are cases where it is needed to map the DTO or entity to two or more classes at the same time. This might be the case for example with JAXB object mapping, data model migration with two separate entity models, or in the case where DTOs are specified in external libraries, may not be edited and thus need to be mapped from within the entity. Additionally, DTO classes might sometimes still be reused or extended in inherited versions. For these purposes the mappings should be defined against mapping with a certain class, or more specifically a type assignable from the actual conversion target or source [(59)]. Since in vast majority of cases the DTO is only mapped with one entity only, the requirement for an explicit definition of the type would create too much overhead. Instead, it could be assumed that if not defined the mapping specified is targeted to mapping with `java.lang.Object`, assignable from all types in Java. This would also enable specifying rules for whole type hierarchies and especially defining mappings with inherited versions as necessary.

In some cases DAO methods return a container object of the entities needed to create DTO for a specific purpose. However, the same DTO could be created directly from the root entity by using its properties to access the related entities. In this case the mapping would be the same for most of the properties in a DTO class with an exception that with the entity container class those properties should be prefixed by the path to the root entity and a few possible changes to inner conversions properties that are now mapped from the container instead of the root entity. In these cases explicitly defining multiple mapping for each property seems like a lot of work especially when the changes are as simple as adding a prefix.

This kind of mapping could be done automatically by allowing a definition of general search paths for all properties in the class [(60)]. Then, if the properties in the entity container would have the same names as those in the entity itself, they could also be automatically mapped. This kind of solution would also help in flattering the class structure into one DTO where some fields come from a related entity. For example, the normalization rules in data model design might produce one-to-one relations such as `LoginUser` and `Person` even though they could be visible to the presentation layer as simply `LoginUserViewDto`.

4.6.3 Customized Conversions

There will always be cases where a generic solution is not applicable for reason or another. For example, some properties of DTOs might not be directly related to an entities: For creation scenario some values might be set as base values or related information fetched by a more complex quires than by the primary key specified in the entity. Some properties might be aggregated, such as count of some related items, sum of invoice rows and so on. Property values might also be formed by joining different columns of an entity such as a name concatenated from first and last name of a person.

For such cases it would be possible to skip the conversion and manually adjust these properties after the actual mapping. However, the mappings this logic relates to might also be part of an inner mapping used in another mapping or collection value conversion. The logic in these adjustments and responsibility of the conversion would hence be easily distributed in different portions of the code base which would result in lowered maintainability and reusability. The mapping component itself should therefore provide an ability to easily customize the logic for a conversion between two given types or override it entirely by user provided implementation in source code level (61). This implementation would then be used in all possible inner conversions for those types without affecting the caller side of the component.

4.6.4 Extendability

Following the Open-Closed Principle and Dependency Inversion Principle of SOLID software design principles [53] and preparing for unknown future requirements, the components of the mapping component should be extendable and changeable. The implementation should allow adding new mapping resolution implementations (62), meaning that should a mapping component only provide mapping by annotations, a mapping by XML could also be added. The existing mapping implementations should also be extendable in a way that for annotation mappings, for instance, it should be possible to introduce new user specified annotations. These custom annotations could be used, for example, to access some repeating concerns in the problem domain.

Also, given that the component offers features such as Hibernate support or Code Generation support, the implementations of these should be encapsulated with interfaces and replaceable (63). This way, should the JPA implementation need to be changed or a code generation library is outdated for the newest Java version, the component could still be used.

5. COMPARION

In this chapter the existing implementations of a generic DTO and entity mapping component are first briefly introduced in Section 5.1 and then evaluated and compared against each other. Based on a comprehensive maturity model presented in Section 5.2 the analysis covers both the functional factors specified in Chapter 4 as well as non-functional factors such as usability, performance and scalability. Finally the overall evaluation results are presented in Section 5.3.

5.1 Implementations

All the implementations of components that can perform runtime mapping of parallel class hierarchies found at the moment of writing this thesis were included in this comparison. The implementations chosen for the comparison are listed in Table 5.1 along with their version and responsible author. Each implementation is described briefly in the following subsections in alphabetical order.

All the components are open-source licensed in a way that they can be used as a part of a commercial software. Only the GNU Lesser General Public License (LGPL), used by Generic DTO Assembler and OMapper, might be problematic in a case where a class of the library would need to be extended by the software because this would make the extending software a derivative work and would need to be licensed under a compatible open-source license. [56]

The different mapping techniques supported by the implementations are listed in Table 5.2. Annotations and mapping API are the two most commonly used with only one implementation offering a purely DSL based mapping.

5.1.1 Dozer

Dozer is the oldest existing implementation of a Java Bean mapping component originating from 2005. It support recursive structures, collection and array conversions and bi-directional mapping. It is designed to work with JAXB and it has built-in support for Spring and some built-in basic type conversions.

Originally Dozer was configurable with XML only but has since added an API for mapping as well as an experimental support for annotation mapping in 2011,

Component	Version	License	Author
Dozer	5.4.0	Apache 2.0	Franz Garsombke, Matt Tierney
Generic DTO Assembler	3.1.0	LGPL	Denis Pavlov
Generic DTO Converter	2.0	MIT	Tommi Ratamaa
jDTO Binder	1.4	Apache 2.0	Juan Alberto Lopez Cavallotti, Gustavo Genovese
JMappper	1.2.0	Apache 2.0	Alessandro Vurro
Modelmapper	0.6.1	Apache 2.0	Jonathan Halterman
Moo	1.3	BSD	Geoffrey Wiseman
Nomin	1.1.1	Apache 2.0	Dmitry Dobrynin
OMapper	2.0	LGPL	Sachin Magician
Orika	1.4.3	Apache 2.0	Matt DeBoer
Spring Object Mapping	1.0.0-SNAPSHOT	Apache 2.0	Rossen Stoyanchev

Table 5.1: Compared implementations.

Component	Annotations	XML	DSL	API	Proxy Objects
Dozer	X	X		X	
Generic DTO Assembler	X			X	
Generic DTO Converter	X			X	X
jDTO Binder	X	X			
JMappper	X	X			
Modelmapper				X	X
Moo	X				
Nomin			X		
OMapper	X				
Orika				X	
Spring Object Mapping				X	

Table 5.2: Mapping techniques supported by the implementations.

which does still, however, cover only a fraction of the features available by XML or API mappings. The latest release is from the end of 2012. [57]

Dozer does not have code generation support, meaning it will use reflection for copying the values. For immutable objects, Dozer does not offer support for custom constructors but will regard the visibility rules by calling a private constructor when necessary. It also supports custom bean factories which additionally make it possible to use Dozer with abstract types and interfaces. [58]

5.1.2 Generic DTO Assembler

Generic DTO Assembler (GeDa) originates from 2009 with the first version published in 2010. It currently has reached a stable phase but still had ongoing development during 2013. GeDA supports recursive bi-directional bean mapping, collection mapping, type conversions and is primarily focused on Entity and DTO mappings. It has integrations for Spring and in 2013 introduced a support for OSGi and multi-class-loader environments. GeDA has support for Annotation and API mappings without type safe proxy objects. [59]

In the desing of GeDA, a lot of effort has been put on performance. It supports a total of three different byte code providers including Javassist, BCEL and Java integrated byte code generator as well as reflection. GeDA is currently the best performing mapping component there is for Java entity and DTO mapping [60]. The use of byte code providers also means that GeDA respects visibility rules.[59]

GeDA also uses bean factories for creating instances of objects with an alias name. This also makes is possible to specify implementations for abstract types and interfaces. However, bean factories in Java code always need to be specified for each mapped type: even default constructors are not used automatically. GeDA does not do automatic mapping: in annotation mapping each mapped class and each mapped field field needs to be marked with annotation and with the API each field pair need to mapped explicitly unless entity and DTO classes are an exact match. Additionally, for every component type used in collections, a specific matcher component need to be created. [59]

GeDA has a support for fetching an entity by primary key by using its `@DtoParent` annotation. For this purpose a specific `EntityRetriever` needs to be defined. Hibernate or other JPA implementation can be used. However, the use of this feature requires the reference in the DTO object to be a bean containing the actual primary key as its property. Additionally, this approach does not work as a solution for fetching collections of related entities. [61]

5.1.3 Generic DTO Converter

Generic DTO Converter (GeDC) project originated from the needs of Dicode and started in the end of 2011 as a free-time project by the author of this thesis work without the knowledge of other existing implementations [62]. It also supports recursive structures, collections and arrays, type conversions and has been designed to be used as a Spring component. GeDC is designed to be extended in terms of mapping techniques, currently supporting mapping by annotations, automatic mapping by matching name and type, camel-case token matching and mapping by API with proxy objects or String paths, or a combination of these techniques.

Version 2.0, released in 2013 introduced a support for dynamic code generation by using Javassist and a wrapping layer, through which other byte code providers could be utilized as well. Before version 2.0, only reflection was used and the most complex parts of the implementation, such as collection mapping, still use reflection. All custom type conversions can be specified in Java code and optionally provide a code generating implementation.

GeDC allows the use immutable objects by specifying custom constructors while and uses default constructors automatically. GeDC was designed to fetch the related entities by primary key as well as collections of related entities by collection or arrays of primary keys, which is not fully supported by any other implementation. JPA annotations are used for specifying the primary keys of the entities. Built-in support is offered for Hibernate only but other JPA implementations could also be used. However, the support is partial as it currently can not use e.g. XML mappings of entities or does not support multiple primary key columns. Automated collection sorting is also implemented.

5.1.4 jDTO Binder

The jDTO Binder was first released in 2011 and has been under development during 2012. It provides bi-directional mapping and mapping over collections and arrays and a wide range of type converters but for recursive mapping, each mapped instance need to be passed explicitly to the interface. The component has support Spring among others and can also be used with Mule Enterprise Service Bus (ESB). jDTO Binder supports XML and annotation based mappings. [63]

The component uses reflection for field access with no dynamic code generation support. While general type conversions are not supported, jDTO Binder supports different kinds of mergers that can also be user defined. For example, one of the built-in mergers supports Groovy scripting expressions to transform property values which allows custom logic to be used in conversions. Immutable objects and custom

constructors are also supported, but the component does not offer any kind of support for fetching entities by primary keys. [63]

5.1.5 JMapper

JMapper was first released in the end of 2012 and current version is from the beginning of 2013. The framework supports recursive structures, arrays, all the collection types and custom logic in the conversions. The mappings can be defined with annotations or XML files which define the mappings for one direction only. Properties can not be mapped using a property path, which makes it impossible to flatter entity aggregations to a single DTO. The framework includes an utility for dynamically modifying the XML mapping files but not an actual API for defining the mappings. [64]

JMapper is based on code generation and uses Javassist as its byte code provider. The custom conversions can either be defined as Java methods in the converted type or as blocks of code in the XML or Java file with replaced placeholders for types and symbol names. These conversions are specified between two given types and used globally, which the framework refers as static conversion, or defined case-by-case as dynamic mappings. Fetching entites by primary keys is not supported. [64]

5.1.6 Modelmapper

Modelmapper was also first released in 2011 and in 2013 is still under development with current version 0.6.1. It supports recursive mappings, type conversions as well as collections and arrays but does require mapping both directions separately. Modelmapper offers a wide range of integrations to other frameworks including Spring. Modelmapper only provides mapping by API but also allows the use of proxy objects and is able to map property pairs automatically based on name and type or camel-case token matches with different strategies. [65]

Modelmapper uses reflection for copying the values. Default constructors are used by default and support for immutable types is achieved by bean factories called `Providers`. Modelmapper also has sophisticated support for generic types and bound checks via its anonymous `TypeToken` but no support for fetching entities by primary keys exists. [65]

5.1.7 Moo

Moo originates from 2009 its latest release version being from 2012 and is currently in a release candidate phase for version 2.0. It supports bi-directional and recursive mappings and collections with synchronization support. Moo uses annotation

mapping with MVEL [66] expression language with similar property mapping features to Groovy. [67]

Moo is based on reflection and runtime interpreted MVEL expressions. It lacks support for immutable objects although private or protected constructors can be called. Moo does not support fetching entities by primary keys automatically but as explained in examples, named references to DAO objects may be passed to MVEL expressions and fetching by primary key can be achieved this way. [67]

5.1.8 Nomin

Nomin has been developed during 2010 and 2011 with no recent releases. The component provides bi-directional mapping capabilities with recursive and collection and array support. Apart from the other implementation the mappings in Nomin are written with Groovy scripts which functioning as a DSL language for the property mappings. Nomin itself is also written in Groovy language which can utilize reflection and can be used from Java applications as it defines Java interfaces and is compiled to JVM byte language. It also has an integration for Spring. [68]

Nomin mappings can either be compiled with the rest of the application as Groovy classes or parsed at runtime from separate script files. Since Grooby is a GPL, mappings can easily be customized. However, a part from the Groovy language, no support for specifying constructors or general type conversions exists, nor does support for JPA or fetching entities by primary keys. [68]

5.1.9 OMapper

OMapper is a simple bean mapping component. It was first released in 2011 and version 2.0 later in 2012. The source codes of the component are not available and documentation is limited to basic use cases which do not cover mapping over collections or arrays nor custom type conversions. Because little information was available, some learning tests were made and rest of the features were assumed non-existing. Mappings are annotation based and limited to one direction only. The interface requires all inner conversion beans to be passed separately with no possibility to construct objects. [69]

5.1.10 Orika

Orika project was started in 2012 and has had multiple releases in 2013. The mapper component supports recursive mappings, custom converters as well as collection and array conversions. The component has an inbuilt Type-system that supports

generics. The mappings can be defined with an API bi-directionally. Orika also provides an integration with the Spring framework. [70]

Orika uses dynamic code generation with Javassist as a byte code provider. Orika supports immutable types with custom converters which may be registered globally or used on case-by-case in basis. Custom constructors can also be specified either in mappings or with a custom `ObjectFactory`. Orika is also highly customizable with its internal parts, including byte code provider, mapping strategies and constructor resolving strategies changeable. Fetching entities by primary keys is, however, not supported. [70]

5.1.11 Spring Object Mapping

Spring Framework has a mapping project founded in 2010 but it is still in 1.0.0 snapshot version meaning that no release versions exists. The component utilizes Spring SpEl language with which, for example makes it possible to build a null-safe reference path can be presented using `?.` operator, filtering conditions may be applied to conversions and collections may be projected [71]. In addition, to SpEL expressions the mapping API provides basic mapping by property names and allows definition of custom converters. [72]

The performance of the component remains under question, since it does not seem to use caches or generated code. Also, for further reading the documentation suggests looking at Dozer, indicating that the component is not intended for the most complex cases. [72]

5.2 Maturity Model

Koljonen (2008) has selected Intel's proposal for Business Readiness Rating for Open Source (OpenBRR) [73] to be used as an assessment model when evaluating an open source software to be taken to use at Dicode Inc [74]. The model consists of multiple categories which are independently evaluated based on sets of criteria. Each of the category results are then scaled and get a score ranging from 1 to 5. The final overall Business Readiness Rating (BRR) is the weighted average of these scores. The categories with their description as in the OpenBRR white paper [73] and the weights chosen for this comparison are shown in Table 5.3.

Functionality and usability are seen as the most important factors for choosing the component since the sole purpose of using such as a component is to ease the work of a developer. The component with higher number of features implemented is more likely to support the mapping in most scenarios with minimized manual work and thus achieve this goal. The functional requirements are covered in the next subsection.

Category	Weight	Description
Functionality	35,0 %	How well will the software meet the average users requirements?
Usability	20,0 %	How good is the UI? How easy to use is the software for end-users? How easy is the software to install, configure, deploy, and maintain?
Quality	10,0 %	Of what quality are the design, the code, and the tests? How complete and error-free are they?
Security	0,0 %	How well does the software handle security issues? How secure is it?
Performance	10,0 %	How well does the software perform?
Scalability	10,0 %	How well does the software scale to a large environment?
Architecture	0,0 %	How well is the software architected? How modular, portable, flexible, extensible, open, and easy to integrate is it?
Support	5,0 %	How well is the software component supported?
Documentation	10,0 %	Of what quality is any documentation for the software?
Acceptance	0,0 %	How well is the component adopted by community, market, and industry?
Community	0,0 %	How active and lively is the community for the software?
Professionalism	0,0 %	What is the level of the professionalism of the development process and of the project organization as a whole?

Table 5.3: Business Process Readiness evaluation categories and their weights with associated OpenBRR descriptions.

Because there is no User-Interface (UI) in the components, usability is defined as the amount of manual work needed to configure the component. This is measured in lines of code as well as number of components and parameters needed to set up both the component and the mappings. Performance and Scalability are also important factors in the web application environment and are discussed further in the following subsections.

On the other hand, security is not the major concern in these kind of components but rather a concern of their environment. When it comes to architecture, very similar solutions were seen in all of the components, and some of the related factors concerning flexibility, extendability or integrations are already included as a part of the functional requirements analysis. In general, all of the components access the same issue and they all cause similar kind of concerns and benefits for the application they are used in. This is way architecture was not included separately in the OpenBRR evaluation.

Most of the components are based on a small community with a limited user-base. This is why factors related to community, acceptance and professionalism are either widely unknown or difficult to measure, and would supposedly result in similar end comes with low margins between the implementations. This is why they are not taken into account as a part of the total result.

Overall, the selection of the 7 categories out of the 12 provided by the OpenBRR is aligned with the suggestion of choosing no more than 7 evaluation categories [73].

5.2.1 Functional Requirements

The functional requirements specified in Chapter 4, were evaluated in detail for each of the compared component implementation. Requirements were rated with importance ranging from 1 to 3 following the guidelines of OpenBRR. Features along with their rating can be found in Appendix A.

Required features were given the importance of 3, nice-to-have features the importance of 2 and less important features the importance of 1. For each component, all features were evaluated based on either the documentation or static code analysis of the component. For OMapper sources were not available so part of the analysis was done based on learning tests. If no signs for the feature was found from the documentation or code, it was considered that the feature does not exist.

Following the OpenBRR, components implementing a feature get as many points as the importance of that feature, zero points if a not implemented feature has the importance of 1 to 2 and -3 points if a required feature was not implemented. For a partial implementation of a feature, less than the maximum points was given. The results of this evaluation is presented in Appendix B. The overall total result for

each component relative to the maximum number of points, 130, was set in to scale from 1 to 5 based on normalization scale presented in Table 5.4.

Normalization scale	**%**	**Result (1-5)**
Excellent (> 96%)	96 %	5
Very good (90% - 96%)	90 %	4
Acceptable (80% - 90%)	80 %	3
Weak (65% - 80%)	65 %	2
Not acceptable (< 65%)	0 %	1

Table 5.4: Normalization scale for functional requirment points.

The normalization scale was chosen as suggested by the OpenBRR with 65 % minimum and 96 % requirement for full points. The percentage was calculated relative to the sum of required functional points, being 72, which means that also negative scores lower than 0 % or higher than 100 % may exist. This puts more weight on the required features and makes it possible to gain extra points for additional features. [73]

Component	**Score**	**%**	**Result (1-5)**
Dozer	66	92	**4**
Generic DTO Assembler	70	97	**5**
Generic DTO Converter	117	163	**5**
jDTO Binder	50	69	**2**
JMapper	45	63	**1**
Modelmapper	52	72	**2**
Moo	32	44	**1**
Nomin	41	57	**1**
OMapper	-38 [1)]	-53	**1**
Orika	75	104	**5**
Spring Object Mapping	32	44	**1**

Table 5.5: Compared implementations. [1)] The existence of some of the features in OMapper could not be verified since no source codes were available.

The total results are shown in Table 5.5 The components having 1 as a result were considered unacceptable. Therefore, JMapper, Moo, Nomin, OMapper and Spring

Object Mapping components were excluded from the rest of the maturity analysis. This exclusion was also required for fair performance and scalability comparisons since with simple cases all the features could not be compared and for more complex scenarios, on the other hand, all the components could not be compared because of the lack of features.

5.2.2 Usability

Usability was evaluated based on the amount of code or configuration needed to initialize the component with an encapsulated instance where mappings are set for both directions. For this comparison, in the higher level the number of necessary classes interfaces and configuration files, annotations was taken into account. Additionally, the number of lines of code without empty lines or comments, and the number of XML elements and attributes, as well as the number of Annotations and their parameters used were calculated.

The weighted totals were then compared to manual implementation in Java code. Because the sole reason for using a mapping component is to reduce the amount of work needed, the weighted total for manual mapping was set as a base value with score of 1. A lower value means less work, and therefore the scores were evenly distributed to the range down from the total for manual mapping. The results and weights are included in Appendix C totals results are presented in Table 5.6.

Component	**Total**	**Result (1-5)**
Java Manual	197	1
Dozer	193	2
Generic DTO Assembler	311	1
Generic DTO Converter	47	5
jDTO Binder	122	3
Modelmapper	88	4
Orika	92	4

Table 5.6: The overall usability results.

The scenario analyzed was based on the DTOs and Entities mapped in the same performance test code used by GeDA [60], the same scenario used later in Performance testing. This way the usability results can directly be compared with performance results, indicating a possible tradeoff where more lower level code and configuration could be used to gain higher performance. The usability relating to

fetching entities by primary keys was not measured because most of the components do not support it.

It can be said, that GeDC, Modelmappera and Orika clearly reduce the amount of work needed for the DTO and Entity mappings. The highest result was achieved by combination of annotations, mapping by matching name and type and camel-case convention based mapping. APIs and automatic mapping by matching name and type were used with Modelmapper and Orika. Mapping by XML was expectedly more verbose than mapping by annotations or an API, with the consequence that mapping with Dozer actually caused approximately as much work as mapping manually.

Interestingly, annotation based GeDA required more lines of code (72 lines) distributed in 4 components than the manual mapping (64 lines) in a single class, and on top of that added 12 annotations with 13 parameters. This is far more than also annotation based GeDC with 8 lines of code in one component, 4 annotations and 2 annotation parameters. The reason for this is the need of specifying a `BeanFactory` for all the class types instantiated, a `DtoToEntityMatcher` for matching the collection elements and an annotation for every DTO class and every field mapped. Considering that both manual and GeDA mappings were written by the developer of the GeDA framework, the configuration overhead can be considered to be as low as it can get, indicating that using GeDA may actually add more work for the developer rather than reduce it.

5.2.3 Performance

Performance tests were based on test cases provided by GeDA [60] with mappings for jDTO Binder and GeDC added. The basic test case included an aggregative mapping of fields of a `Person` class to `PersonDTO` and vise versa with flatterning mapping of `Name` and inner conversion of `ContactAddress` and flattering of its `Country` object. Second test case included inheritance and an additional mapping over collection of `ContactAddress`es. Both scenarios were run for both directions, from Entity to DTO and from DTO to Entity, and tested with 100 and 10 000 mappings for each scenario. The different number of mappings were used to validate whether the results are linear.

Because mapping manually in Java code is the theoretical performance maximum for the test cases, manual mapping was also tested and the runtime was used as a base value against which to compare the performance of the components. The comparisons were done in logarithmic scale as described in Equation 5.1. In the equation α was used as a scaling factor for the manual mapping runtime. In the basic scenario, it is likely that JIT compiler was able to optimize the manual mapping

so that it is not directly comparable and thus a scaling of $\alpha = 40$ was used. In the complex scenario collection related operations ramped the manual runtime, so that direct comparison with $\alpha = 1$ could be used. This score is limited to the range from 1 to 5. The overall results are rounded from the average of all the test cases.

$$score = 5 - \alpha log_{10}(t_{manual}/ns) + log_{10}(t_{measured}/ns) \tag{5.1}$$

Tests were run with Caliper microbenchmark framework which runs the tests multiple times until the deviation settles, trying to minimizes the impact of for example warm-ups and JIT compiler optimization [75]. The test results along with graphs for each scenario are shown in Appendix D. The overall results as averages for the test cases are shown in Table 5.7.

Component	**Basic Score**	**Collection Score**	**Result (1-5)**
Dozer	2.4	2.4	2
Generic DTO Assembler	3.9	3.9	4
Generic DTO Converter	3.6	3.0	3
jDTO Binder	3.1	3.4	3
Modelmapper	3.0	3.1	3
Orika	4.1	4.3	4

Table 5.7: The overall performance test results.

The results are in line with the performance results of GeDA [60] as well as test results collected by Anatoliy Sokolenko with Dozer, Orika and Manual mapping compared [76]. From the results, we can conclude that, as expected, all of the components perform linearly when comparing the runtimes of scenarios with 100 and 10 000 mappings. It is also notable that components using dynamic code generation perform roughly ten times as fast as the components using only reflection.

Overall, the performance for all of the components is in reasonable range. In practice, none of the components would cause a bottleneck in the web application where most of the time is spent waiting database queries. The number of mapped objects during one request is rarely over 100 since for queries that need better performance objects are often hydrated in the persistence layer directly. For the more complex aggregation, converting 100 instances took less than 1 ms for most of the components and around 2,5 ms with the slowest implementation, which is still likely less than the database query would take.

5.2.4 Scalability

For scalability tests, a Java EE Servlet environment was used. The environment was intentionally as thin as possible, with no other framework than the standard Java EE Servlet version 2.5 API with a scalable Jetty 6.10 [77] Servlet container with minimal overhead used. For each of the components, a dedicated Servlet was used with an initialized singleton reference to the corresponding mapping component. Each request to the Servlet performed 10 000 mappings with a scenario equal to the basic case used in performance tests. The tests were carried out with Apache JMeter [78] version 2.10 test tool over HTTP protocol against a Jetty server running on localhost with no delays caused by the network. For comparison, a Servlet performing a manual mapping was also included.

Tests were first carried out with a single mapping operation and with 100 mapping operations, but these did not show any measurable differences between the components. With 1000 concurrent threads and a ramp-up period of 10 seconds, every Servlet could respond with less than 1 ms average response time for a single mapping. With 2000 concurrent requests, on the other hand, the environment limits were met and results showed nothing but noise with high deviations between each round. This along should indicate, that normal use of the components will not cause a bottleneck for the application that typically includes more overhead caused by the Servlet container, framework and especially the persistence layer

An intentionally non-realistic number of 10 000 mappings per requests with 1000 concurrent threads over 20 seconds shows a difference in scaling. The tests were carried out 3 times and averages were calculated. Results are presented in Appendix E. The results for these setting show three kinds of behavior which can be observed in graphs presented in Appendix F. Manual mapping, GeDA and Orika responded with almost constant throughput through the whole test indicating that the actual throughput would be more than the 30 000 request performed, while for Modelmapper and GeDC the throughput was increasing linearly. For Dozer and jDTO Binder the throughput decreased rapidly and finally reached the denial of service state where server connections run out and errors were included in the results. This is why for them, medium hit the maximum 30 seconds.

In the results the measured throughput was scaled from 1 to 5 relatively to the manual throughput and 1 result point was reduced if the results showed errors indicating a denial of service state. The results are shown in Table 5.8.

The results are in line with the performance results except for jDTO Binder which outperformed Modelmapper in performance analysis but was now at the same

Component	Average throughput	Errors	Result (1-5)
Dozer	1161 req/min	X	2
Generic DTO Assembler	2988 req/min		5
Generic DTO Converter	1649 req/min		3
jDTO Binder	1160 req/min	X	2
Modelmapper	1372 req/min		3
Orika	2987 req/min		5

Table 5.8: The overall scalability results.

throughput as the slowest implementation, Dozer, in performance tests. This could indicate a potential scalability problem with jDTO Binder.

5.2.5 Other Categories

Other chosen categories in the Business Process Readiness included Quality, Documentation and Support. The criterion for these categories were selected based on OpenBRR recommendations [73]. However, especially the requirements for number of releases, reported bugs and number of messages in mailing lists were reduced from the recommended values because the components do not expectantly have a wide user community. The professionalism in Support section was replaced by number of StackOverflow questions and answers. The selected criterion and measures for results are listed in Appendix G. The evaluations for each component are shown in Appendix H.

Component	Qality	Support	Documentation
Dozer	4	4	4
Generic DTO Assembler	3	2	3
Generic DTO Converter	4	1	2
jDTO Binder	4	3	3
Modelmapper	3	4	3
Orika	3	4	3

Table 5.9: The overall results for other categories.

The overall results for these categories are listed in Table 5.9. The measures selected did not show significant differences between the components other than

relatively poor results for support and documentation for Generic DTO Converter. However, the total weight for these categories in the overall BRR rating is only 25

As the support was measured by the number of messages in mailing lists, forums and StackOverflow service, it prefers the components with wider user community. The quality measures were strongly based on the number of issues and releases in a given period of time where a high number is generally considered worse than low. A component with wider user space and thus more issues reported and quick fixes needed, might therefore have a disadvantage in this scale. When it comes to documentation, except for Dozer having the documentation included in source codes, other frameworks provided no framework for the users to extend the documentation.

5.3 Evaluation of Results

The total Business Readiness Ratings for the OpenBRR analysis are presented in Appendix I for each of the anlayzed component as sums of weighted scores for each analyzed category. The cumulative BRRs are listed in Table 5.10 ordered by the rating.

Component	**BRR**
Orika	4.3
Generic DTO Converter	4.0
Generic DTO Assembler	3.6
Dozer	3.2
Modelmapper	2.9
jDTO Binder	2.7

Table 5.10: The total Business Readiness Ratings.

Based on the ratings it can be said that there exists implementations of a generic DTO and entity mapper component ready to be used for business. This evaluation was based for the needs of Dicode Ltd. To satisfy those needs, Orika and Generic DTO Converter are both very good solutions with Orika having the highest overall rating. The usability results reduced the score of Generic DTO Assembler to only acceptable level along with Dozer.

6. CONCLUSIONS

The goal for this thesis was to find a generic solution to support or to automate entity and DTO mapping performed in between the application layers of a Java EE web application. For Dicode Ltd. the absence of such a solution previously caused overhead both in coding and maintenance phase of the application as well as architectural concerns such as shift in the responsibility of components and the reuse of DTO classes.

In this thesis several approaches were first recognized as possibilities for the overall mapping process, including the use of a IDE plugin, mapping generation with a build automation tool or a runtime mapping component. Based on existing implementations and because of the dynamic nature of the problem, JPA mappings in especial, a runtime component approach was chosen and further analyzed.

The runtime component may use different mapping techniques such as XML, annotations, API mapping with or without proxy objects or mapping with DSL, all having their advantages and disadvantages in terms of overhead caused and stabilizability they provide. Patterns such as Early Work and Lazy Execution Patterns along with Caches can be used to gain performance advantages. While the actual mapping is based on reflection, the actual copying of values may be performed more efficiently using dynamic code generation.

Requirements for a runtime mapping component were recognized and analyzed based on the needs of Dicode Ltd. The component should integrate with currently used techniques and frameworks such as Spring, Hibernate and Maven but, at the same, be extendable enough to allow support for future changes. For a general purpose component, there exist many requirements to cover the most common use scenarios, but it should be noted that all needs can never be fully covered and thus the component should allow custom mappings and conversions. Above all, the component and mapping techniques used should cause minimal overhead to the developer in terms of configuration since that was the original reason for the use of such a component.

Eleven open source components were recognized as possible solutions for the problem. For them, a comprehensive Business Process Readiness analysis including total of seven weighted categories was performed. In the first stage, components were compared in the most highly weighted feature category, and based on the

results, five components were ranked out of the comparison. For the rest six components, usability, quality, performance, scalability, support and documentation were analyzed.

In the second most weighted category, usability, measured in the amount of configuration, two components received an unacceptable or a bad score, meaning that they cause more configuration or coding overhead than manual mapping in Java code in the measured scenario. On the other hand, three of the components caused a relative overhead less than half compared to the manual mapping, while providing flexibility for future changes and null-safety at the same time. Generic DTO Converter caused least overhead cutting the manual overhead to less than one fourth.

One of the important features would have been integration with Hibernate or other JPA implementation to automatically fetch related entities by their primary keys in the corresponding entity objects. Unfortunately, however, only Generic DTO Converter implemented the feature even partially. Most of the compared implementation did not implement this feature, and thus the usability related aspects of could therefor not be compared.

When it comes to performance and scalability, all results were acceptable but the components using dynamic code generation including especially Orika and Generic DTO Assembler, were measured to both perform and scale better than the rest. However, based on the scalability tests, no actual impacts could be measured for a real web application with a reasonable number of mappings.

In the overall BRR ratings, two components, Orika and Generic DTO Converter were above others achieving ratings in very good range, Orika having the highest weighted score, 4.3 out of 5. This implicates that there exists generic solutions for entity and DTO mapping based on the requirements of Dicode Ltd. Furthermore, based on the analysis, the use of either of the two implementation can be recommended for future projects.

REFERENCES

[1] Dhrubojyoti, K. *Pro Java EE Spring Patterns: Best Practices and Design Strategies Implementing Java EE Patterns with the Spring Framework*. Apress, USA, 2008. ISBN 978-1-4302-1009-2.

[2] Keith, M. and Schincariol, M. *Pro JPA 2: Mastering the Java Persistence API*. Apress, USA, 2009. ISBN 978-1-4302-1956-9. URL `http://library.books24x7.com/assetviewer.aspx?bookid=33335&chunkid=1&rowid=2`. Referenced 21/10/2013.

[3] Johnson, R., Hoeller, J., and Donald, K. e. Spring Framework Reference Documentation, 2013. URL `http://docs.spring.io/spring/docs/3.2.4.RELEASE/spring-framework-reference/pdf/spring-framework-reference.pdf`. Referenced 21/10/2013.

[4] King, G. and Muir, P. e. a. JBoss Web Framework Kit 2.1 - seam reference guide for use with JBoss Web Framework Kit 2, 2013. URL `https://access.redhat.com/knowledge/docs/en-US/JBoss_Web_Framework_Kit/2.1/html-single/Seam_Reference_Guide/index.html`. Referenced 24/03/2013.

[5] Sun Microsystems. JSR 317: Java Persistence API, Version 2.0, November 2009. URL `http://download.oracle.com/otndocs/jcp/persistence-2.0-fr-eval-oth-JSpec/`. Referenced 01/03/2013.

[6] Hat, R. HIBERNATE - Relational Persistence for Idiomatic Java, March 2013. URL `http://docs.jboss.org/hibernate/orm/4.1/manual/en-US/html_single/`. Referenced 24/03/2013.

[7] Oracle. Oracle Fusion Middleware Developer's Guide for Oracle TopLink 11g Release 1, 2013. URL `http://docs.oracle.com/cd/E14571_01/web.1111/b32441/undtl.htm#CHDGCDDB`. Referenced 24/03/2013.

[8] Understanding EclipseLink - 2.4, March 2013. URL `http://www.eclipse.org/eclipselink/documentation/2.4/eclipselink_otlcg.pdf`. Referenced 24/03/2013.

[9] Batoo JPA Documentation, 2013. URL `http://batoo.jp/documentation/`. Referenced 24/03/2013.

[10] Apache. Apache OpenJPA 2.2 User's Guide, Oct 2012. URL `http://openjpa.apache.org/builds/2.2.1/apache-openjpa/docs/manual.pdf`. Referenced 24/03/2013.

[11] Oracle. Java Platform, Enterprise Edition, v 5.0 API Specifications, 2007. URL `http://docs.oracle.com/javaee/5/api/`. Referenced 3/12/2013.

[12] Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R., and Stafford, R. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002. ISBN 0-321-12742-0.

[13] Microsoft. Data Transfer Object, 2013. URL `http://msdn.microsoft.com/en-us/library/ff649585.aspx`. Referenced 16/03/2013.

[14] Esposito, D. Pros and Cons of Data Transfer Objects. *MSDN Magazine*, August 2009. URL `http://msdn.microsoft.com/en-us/magazine/ee236638.aspx`. Referenced 16/02/2013.

[15] Kawaguchi, K., Vajjhala, S., and Fialli, J. The Java®Architecture for XML Binding (JAXB) 2.1, Dec 2006. URL `http://download.oracle.com/otndocs/jcp/jaxb-2.1-mrel-eval-oth-JSpec/`. Referenced 24/03/2013.

[16] Hardan, R. Modelbridge - Java Object Mapping, 2013. URL `http://www.modelbridge.org/`. Referenced 20/10/2013.

[17] Bray, T. and Paoli, J. e. Extensible Markup Language (XML) 1.0 (Fifth edition), Nov 2008. URL `http://www.w3.org/TR/REC-xml/`. Referenced 18/10/2013.

[18] Oracle. JavaTM 2 Platform Standard Edition 5.0 - Annotations, 2010. URL `http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html`. Referenced 15/09/2013.

[19] Oracle. Java Platform, Standard Edition 7 API Specification, 2013. URL `http://docs.oracle.com/javase/7/docs/api/`. Referenced 12/10/2013.

[20] Partington, V. JPA Implementation Patterns: Lazy Loading, Aug 2009. URL `http://java.dzone.com/articles/jpa-lazy-loading`. Referenced 20/10/2013.

[21] Darcy, J. D. JEP 126: Lambda Expressions Virtual Extension Methods, Feb 2013. URL `http://openjdk.java.net/jeps/126`. Referenced 20/10/2013.

[22] Fusco, M. lambdaj, 2013. URL `http://code.google.com/p/lambdaj/`. Referenced 20/10/2013.

[23] Mernik, M., Heering, J., and Sloane, A. M. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37:316–344, Dec 2005.

[24] Colebourne, S. Enhanced null handling - Invocation and defaulting, 2013. URL `https://docs.google.com/document/d/1hX_krmhniT2PT7hdTDMFsds7EiC55wsEuR3QZ8UZ-B4/pub`. Referenced 18/10/2013.

[25] Xtext. Xtext 2.4.3 Documentation, Sept 2013. URL `http://www.eclipse.org/Xtext/documentation/2.4.3/Documentation.pdf`. Referenced 18/10/2013.

[26] Neale, M. Groovy - A dynamic language for the Java platform, Sep 2013. URL `http://docs.codehaus.org/display/GROOVY/Home`. Referenced 20/10/2013.

[27] Schinz, M. and Haller, P. A Scala Tutorial for Java programmers, Oct 2013. URL `http://www.scala-lang.org/docu/files/ScalaTutorial.pdf`. Referenced 20/10/2013.

[28] Gabriele, J. Introduction to Clojure, 2013. URL `http://clojure-doc.org/articles/tutorials/introduction.html`. Referenced 20/10/2013.

[29] Oracle. java.lang.reflect API documentation (Java Platform SE 7), 2013. URL `http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/package-summary.html`. Referenced 15/09/2013.

[30] Sosnoski, D. Java programming dynamics, Part 2: Introducing reflection, Juny 2003. URL `http://www.ibm.com/developerworks/library/j-dyn0603/`. Referenced 09/04/2013.

[31] Crisostomo, E. M. Java Generics Tutorial, March 2011. URL `http://thegreyblog.blogspot.fi/2011/03/java-generics-tutorial-part-i-basics.html`. Referenced 15/09/2013.

[32] Oracle. Java 2 SDK, Standard Edition Documentation - version 1.4.2, 2010. URL `http://docs.oracle.com/javase/1.4.2/docs/api/`. Referenced 3/12/2013.

[33] Jenkov, J. Java Reflection: Generics, 2013. URL `http://tutorials.jenkov.com/java-reflection/generics.html`. Referenced 15/09/2013.

[34] Oracle. Java 2 SDK, Standard Edition Documentation - version 1.3.1, 2010. URL `http://docs.oracle.com/javase/1.3/docs/api/`. Referenced 12/10/2013.

[35] Oracle. The Java HotSpot Performance Engine Architecture, 2013. URL `http://www.oracle.com/technetwork/java/whitepaper-135217.html`. Referenced 15/09/2013.

[36] Buzdin, D. Is Java Reflection Really Slow?, Jan 2011. URL `http://www.buzdin.lv/2011/01/is-java-reflection-really-slow.html`. Referenced 15/09/2013.

[37] Würthinger, T., Wimmer, C., and Stadler, L. e. Dynamic Code Evolution for Java. 2013. URL `http://ssw.jku.at/Research/Papers/Wuerthinger10a/Wuerthinger10a.pdf`. Referenced 18/10/2013.

[38] Rose, J. and Würthinger, T. HotSwap - Da Vinci Machine Project - Oracle, 2013. URL `https://wikis.oracle.com/display/mlvm/HotSwap`. Referenced 18/10/2013.

[39] ZeroTurnaround. What developers want: The End of Application Redeploys, 2013. URL `http://files.zeroturnaround.com/pdf/JRebelWhitePaper2012-1.pdf`. Referenced 05/10/2013.

[40] ZeroTurnaround. JRebel Frequently Asked Questions - How does JRebel work?, 2013. URL `http://zeroturnaround.com/software/jrebel/learn/faq/#2`. Referenced 05/10/2013.

[41] ZeroTurnaround. JRebel Plugins, 2013. URL `http://zeroturnaround.com/software/jrebel/learn/jrebel-plugins/`. Referenced 05/10/2013.

[42] Bloch, J. *Effective Java - Second Edition*. Addison-Wesley, California, USA, 2008. ISBN 0-321-35668-3.

[43] Sosnoski, D. Java programming dynamics, Part 8: Replacing reflection with code generation, Jun 2004. URL `http://www.ibm.com/developerworks/library/j-dyn0610/`. Referenced 20/10/2013.

[44] Pavlov, D. Method Synthesizers - GeDA - Generic DTO Assembler, May 2013. URL `http://www.inspire-software.com/confluence/display/GeDA/Method+Synthesizers`. Referenced 18/10/2013.

[45] Cglib. Cglib - Code Generation Library, 2004. URL http://cglib.sourceforge.net/. Referenced 20/10/2013.

[46] Foundation, T. A. S. Apache Commons - BCEL, Oct 2011. URL http://commons.apache.org/proper/commons-bcel/. Referenced 20/10/2013.

[47] Chiba, S. Javassist, Jun 2013. URL http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/. Referenced 20/10/2013.

[48] Bruneton, E. ASM 4.0 - A Java bytecode engineering library, 2011. URL http://download.forge.objectweb.org/asm/asm4-guide.pdf. Referenced 20/10/2013.

[49] Oracle. Java Platform, Enterprise Edition, 6 API Specification, 2011. URL http://docs.oracle.com/javaee/6/api/. Referenced 3/12/2013.

[50] Höller, J. Spring 4 on Java 8 - A Work in Progress, Jul 2013. URL http://www.slideshare.net/ZeroTurnaround/juergen-hoellerspring4onjava8-24613140. Referenced 20/10/2013.

[51] Ebersole, S. Deprecated CGLIB support, Aug 2010. URL http://relation.to/16658.lace. Referenced 20/10/2013.

[52] Yu, B. W. Scaling Your Java EE Applications, July 2008. URL http://www.theserverside.com/news/1363681/Scaling-Your-Java-EE-Applications. Referenced 09/03/2013.

[53] Stenberg, J. SOLID Design Principles and Other Patterns Revisited For .NET, Aug 2013. URL http://www.infoq.com/news/2013/08/solid-principles-revisited. Referenced 20/10/2013.

[54] Hamilton, G. JavaBeans, August 1997. URL http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/. Referenced 09/04/2013.

[55] Fowler, M. POJO, 2013. URL http://www.martinfowler.com/bliki/POJO.html. Referenced 09/04/2013.

[56] Free Software Foundation. GNU Lesser General Public License, June 2007. URL http://www.gnu.org/copyleft/lesser.html. Referenced 27/10/2013.

[57] dozer. Dozer - About, 2012. URL http://dozer.sourceforge.net/documentation/about.html. Referenced 04/11/2013.

[58] dozer. Dozer - Frequently Asked Questions, 2012. URL http://dozer.sourceforge.net/documentation/faq.html. Referenced 04/11/2013.

[59] Pavlov, D. Generic DTO Assembler - Getting started, 2013. URL http://www.inspire-software.com/confluence/display/GeDA/Getting+started. Referenced 04/11/2013.

[60] Pavlov, D. Generic DTO Assembler - Benchmarks, 2013. URL http://www.inspire-software.com/confluence/display/GeDA/Benchmarks. Referenced 04/11/2013.

[61] Pavlov, D. Generic DTO Assembler - DTO parent, 2011. URL http://www.inspire-software.com/confluence/display/GeDA/DTO+parent. Referenced 04/11/2013.

[62] Ratamaa, T. Generic DTO Converter, 2013. URL http://ryh.dy.fi/trac/dtoconverter. Referenced 05/11/2013.

[63] Cavallotti, J. A. L. jDTO Binder 1.4 User's Guide, 2013. URL https://github.com/jDTOBinder/jDTO-Binder/raw/master/book/jdto.pdf. Referenced 04/11/2013.

[64] Vurro, A. jmapper-framework, 2013. URL http://code.google.com/p/jmapper-framework/wiki/Introduction. Referenced 10/11/2013.

[65] Halterman, J. ModelMapper - User Manual, 2013. URL http://modelmapper.org/user-manual/. Referenced 04/11/2013.

[66] Brock, M. MVEL, 2012. URL http://mvel.codehaus.org/. Referenced 04/11/2013.

[67] Wiseman, G. Moo - User Guide, 2013. URL https://github.com/geoffreywiseman/Moo/wiki/User-Guide. Referenced 04/11/2013.

[68] Dobrynin, D. Introduction to the Nomin Java Mapping Framework, 2011. URL http://nomin.sourceforge.net/. Referenced 04/11/2013.

[69] Magician, S. Introduction and usage guide for OMapper, 2011. URL http://code.google.com/p/omapper/wiki/Home. Referenced 05/11/2013.

[70] DeBoer, M. Orika - User Guide, 2013. URL http://orika-mapper.github.io/orika-docs/. Referenced 10/11/2013.

[71] Spring Expression Language (SpEL), 2010. URL http://docs.spring.io/spring/docs/3.0.x/reference/expressions.html. Referenced 10/11/2013.

[72] Spring Foundation. Spring 3 Object Mapping, 2010. URL http://docs.spring.io/spring/previews/mapping.html. Referenced 10/11/2013.

[73] OpenBRR.org. Business Readiness Rating for Open Source, 2005. URL http://docencia.etsit.urjc.es/moodle/mod/resource/view.php?id=4343. Referenced 21/10/2013.

[74] Koljonen, M. Avoimen lähdekoodin valintakriteerit. Diplomity, Tampereen teknillinen yliopisto, May 2007.

[75] Caliper. Caliper - Microbenchmarking framework for Java, 2013. URL https://code.google.com/p/caliper/wiki/JavaMicrobenchmarks. Referenced 10/11/2013.

[76] Sokolenko, A. Dozer vs Orika vs Manual, 2013. URL http://blog.sokolenko.me/2013/05/dozer-vs-orika-vs-manual.html. Referenced 10/11/2013.

[77] The Eclipse Foundation. Jetty - Servlet Engine and Http Server, 2013. URL http://www.eclipse.org/jetty/. Referenced 10/11/2013.

[78] Apache Software Foundation. Apache JMeter, 2013. URL http://jmeter.apache.org/. Referenced 10/11/2013.

A. APPENDIX: FUNCTIONAL REQUIREMENTS

Functional requirements for a generic entity and DTO mapping component

#	Requirement	Importance (1-3)
Maven support		
1	Available as a Maven artifact	**3**
2	Availability in Maven main repository	1
Spring support		
3	Can be used as a Spring component	**3**
4	Customizable to be used in other web frameworks	2
5	Thread-safety	**3**
6	Extendable interface for Spring component	2
JPA support		
7	Fetching entities by primary keys	**3**
8	Fetching multiple entities by array/collection of primary keys	**3**
9	Persistence depended part as a separate Maven artifact	1
10	Hibernate compliant implementation	**3**
11	Support for any JPA implementation	2
Mapping techniques		
12	Annotation-driven configuration	**3**
13	Automaitc mapping by name and type	**3**
14	Automatic mapping by camel case naming convention	2
15	Mapping by XML	2
16	Mapping by DSL	1
17	Mapping by API	1
18	Mapping type safely with proxy objects	2
19	Conditional mapping	1
Bi-directional mapping		
20	Mapping once for both directions	**3**
21	Preventing mapping for either direction	**3**
22	Determing the conversion direction by getter/setter visibility	2
23	Defining annotation mappings to either source or target class	2
Aggregation mapping		
24	Mapping by property paths	**3**
25	Null-safe access	**3**
26	Fetching container object by primary key	2
27	Automatic creation of container objects	2
28	Inner conversion	**3**
29	Mapping inner conversions from container by annotations	1
Type Support		
30	Concrete implementation type can be specified for an abstract type	2
31	Type aliases for abstract types or interfaces	2
32	General default implementation types for abstract types	2
33	Generics type support	2
Type conversions		
34	Ability to add new conversions	**3**
35	Automatic conversion between primary and boxed types	2
36	Ability to replace the conversions	2
37	Null-safety with primitives	1
38	Case-by-case type conversion	1
39	General type convesions	**3**

Collection and array support		
40	Inner conversion over collections and arrays	**3**
41	Automatic conversion between Collection implementations and arrays	2
42	Determing Collection types automatically by Java's reflection	**3**
43	Ordering of DTOs by specifying a Comparator	2
44	Ordering by annnotations specifying order by properties and directions	1
45	Synchronization by primary key mapped property	2
46	Synchronization: adding new and removing non existing	2
47	Collection projection	1
48	Filtering collection values	2
Field and getter/setter support		
49	Support for both getters/setters and fields	**3**
50	Respecting visibility rules	**3**
51	Appliying annotations to eihter field or getter	1
Immutable object support		
52	Support for custom constructors	**3**
53	Ability to specify which constructor to use with the conversion	**3**
54	Defining annotations from the container	1
55	Parameter mappings with all the same features as properties	2
Support for graphs and two-way linking structures		
56	Support with hierarchical structures with parent references	1
57	Supoort for nets and complex structures with reference cache	2
Customizability		
58	Skipping certain property mappings	**3**
59	Multiple mappings by conversion source/targe type	**3**
60	Search paths for property mappings	1
61	Customized conversions	**3**
62	New mapping resolution implementations can be added and extended	1
63	JPA and Code Gernation components should encapsulated and changeable	1
	Total	134
	Total for required features	**72**

B. APPENDIX: FUNCTIONAL REQUIREMENTS EVALUATION

Functional Requirements Evaluation

#	*Requirement*	*Importanc*	*Implementation*									
Maven support			**Dozer**	**GeDA**	**GeDC**	**jDTO**	**JMap.**	**Modlm.**	**Moo**	**Nomin**	**OMap.**	**Orika**	**Spring**
1	Maven Artifact	3	3	3	3	3	3	3	3	3	-3	3	2[5)]
2	In Maven Central	1	1	1	0	1	1	1	1	1	0	1	1
Spring support													
3	Spring Support	3	3	3	1[1)]	3	1[1)]	3	1[1)]	3	1[1)]	3	3
4	Other Framework Support	2	2	2	1[1)]	2	1[1)]	2	1[1)]	2	0	1[1)]	0
5	Thread-safety	3	3	3	3	3	3	3	3	3	-3	3	3
6	Interfaced mapper	2	2	2	2	2	2	0	0	2	0	2	2
JPA support													
7	Fetching by primary keys	3	-3	2[2)]	2[2)]	-3	-3	-3	1[1)]	-3	-3	-3	-3
8	Collection of primary keys	3	-3	-3	3	-3	-3	-3	-3	-3	-3	-3	-3
9	Separate Maven artifact	1	0	0	1	0	0	0	0	0	0	0	0
10	Hibernate Support	3	-3	2[1)]	3	-3	-3	-3	-3	-3	-3	-3	-3
11	Support for any JPA	2	0	1[1)]	1[1)]	0	0	0	0	0	0	0	0
Mapping techniques													
12	Annotation mapping	3	2[6)]	3	3	3	3	-3	3	-3	3	-3	-3
13	Name and type mapping	3	3	3	3	-3	3	3	3	3	-3	3	3
14	Camel case mapping	2	0	0	2	0	0	2	0	0	0	0	0
15	Mapping by XML	2	2	0	0	2	2	0	0	0	0	0	0
16	Mapping by DSL	1	0	0	0	0	0	0	0	1	0	0	0
17	Mapping by API	1	1	1	1	0	0	1	0	0	0	1	1
18	Mapping with proxy objects	2	0	0	2	0	0	2	0	0	0	0	0
19	Conditional mapping	1	0	0	0	0	0	1	0	1	0	0	1
Bi-directional mapping													
20	One configuration for both w	3	3	3	3	3	-3	-3	3	3	3	3	-3
21	Preventing mapping directio	3	3	3	3	-3	2[1)]	2[1)]	-3	-3	3	3	-3
22	Directions by getter/setter v	2	2	2	2	2	0	2	2	2	0	2	0
23	Annotations in source/target	2	2	0	2	2	0	0	2	0	2	0	0
Aggregation mapping													
24	Mapping by property paths	3	3	3	3	3	-3	3	3	3	-3	3	-3
25	Null-safe access	3	3	3	3	3	3	3	3	3	-3	3	3
26	Fetching container object by	2	0	0	2	0	0	0	0	0	0	0	0
27	Creation of container object	2	2	2	2	2	2	2	2	2	0	2	2
28	Inner conversion	3	3	3	3	3	3	3	3	3	-3	3	3
29	Inner conversions mapping f	1	0	0	1	0	0	0	0	0	0	0	0
Type Support													
30	Specify implementation type	2	2	2	2	0	0	0	0	0	2	2	0
31	Type aliases	2	0	2	2	0	0	0	0	0	0	0	0
32	Default implementation type	2	0	2	2	0	2	2	0	0	0	2	2
33	Generics type support	2	0	0	0	0	0	2	0	2	0	2	0
Type conversions													
34	Ability to add new conversio	3	3	3	3	3	3	3	-3	-3	-3	3	3
35	Primary and boxed types cor	2	2	2	2	2	2	2	2	2	2	2	2
36	Ability to replace the conver	2	2	2	2	2	2	2	0	0	0	2	2
37	Null-safety with primitives	1	1	1	1	1	1	1	0	1	1	1	1
38	Case-by-case type conversio	1	1	1	0	1	1	1	0	1	0	1	1
39	General type convesions	3	3	-3	3	-3	3	3	-3	3	-3	3	3

	Collection and array support		Dozer	GeDA	GeDC	jDTO	JMap.	Modelm	Moo	Nomin	Omap.	Orika	Spring
40	Inner conversion	3	3	3	3	3	3	3	3	3	-3	3	3
41	Collection and arrays conver	2	2	2	2	2	2	2	2	0	0	2	2
42	Generic contained type dete	3	3	3	3	3	3	3	-3	3	-3	3	-3
43	Ordering by Comparator	2	0	0	2	0	0	0	0	2	0	0	0
44	Ordering by annnotations	1	0	0	1	0	0	0	0	0	0	0	0
45	Synchronization by primary I	2	0	1[1)]	1[3)]	0	0	0	1[1)]	0	0	0	0
46	Add/remove-merge	2	2	2	1[3)]	0	0	1[4)]	2	2	0	1[3)]	0
47	Collection projection	1	0	0	1	0	0	0	0	0	0	0	1
48	Filtering collection values	2	0	0	0	0	0	0	0	1	0	0	1
	Field and getter/setter support												
49	Getters/setters and fields	3	-3	-3	3	3	3	-3	3	-3	-3	3	3
50	Respecting visibility rules	3	3	3	3	3	3	3	3	3	-3	3	3
51	Annotations to field/getter	1	0	1	1	1	0	0	1	0	0	1	0
	Immutable object support												
52	Support for cconstructors	3	1[1)]	1[1)]	3	3	-3	-3	-3	-3	-3	1[1)]	-3
53	Specify constructor to use	3	-3	-3	3	3	-3	-3	-3	-3	-3	3	-3
54	Annotations from the contai	1	0	0	1	0	0	0	0	0	0	0	0
55	Parameter mapping convers	2	0	0	2	2	0	0	0	0	0	0	0
	Support linked structures												
56	Parent references	1	0	0	1	0	0	0	0	0	0	0	0
57	Reference cache	2	2	2	2	2	0	2	2	2	0	2	2
	Customizability												
58	Skipping mappings	3	2[1)]	3	3	2[1)]	3	3	3	2[1)]	2[1)]	3	3
59	Multiple mappings	3	3	-3	3	-3	3	3	-3	3	-3	3	3
60	Search paths	1	0	0	1	1	0	1	0	0	0	0	0
61	Customized conversions	3	3	3	3	-3	3	3	3	3	-3	3	3
62	Resolution extendability	1	0	0	1	0	0	0	0	0	0	1	0
63	Subcomponents changeable	1	0	1	1	0	0	0	0	0	0	1	0
	Totals	**72**	**66**	**70**	**117**	**50**	**45**	**52**	32	41	-38	**75**	32

[1)] = No specialized support but can be used.

[2)] = Does not support multiple primary keys for an entity.

[3)] = No element mathcer, based on order.

[4)] = No matching or removing

[5)] = Not separated.

[6)] = Experimental, limited feature

Normalization scale	%	Result (1-5)
Excellent (> 96%)	96 %	5
Very good (90% - 96%)	90 %	4
Acceptable (80% - 90%)	80 %	3
Bad (65% - 80%)	65 %	2
Unacceptable (< 65%)	0 %	1

Implementation	Component and version	Score	%	Result (1-5)
Dozer	**Dozer 5.4.0**	**66**	**92 %**	**4**
GeDA	**Generic DTO Assembler 3.1.0**	**70**	**97 %**	**5**
GeDC	**Generic DTO Converter 2.0**	**117**	**163 %**	**5**
jDTO	**jDTO Binder 1.4**	**50**	**69 %**	2
JMap.	JMapper 1.2.0	45	63 %	1
Modlm.	**Modelmapper 0.6.1**	**52**	**72 %**	2
Moo	Moo 1.3	32	44 %	1
Nomin	Nomin 1.1.1	41	57 %	1
OMap.	OMapper 2.0	-38	-53 %	1
Orika	**Orika 1.4.3**	**75**	**104 %**	**5**
Spring	Spring Object Mapping 1.0.0-SNAPSHOT	32	44 %	1

C. APPENDIX: USABILITY TEST RESULTS

Usability Test Results

Appr.	Factor	Weigth
NoCl	Number of classes or interfaces	5
NoCF	Number of configuration files	5
LoC	Lines of Code	3
NoXE	Number of XML elements	3
NoXA	Number of XML attributes	3
NoA	Number of Annotations	3
NoAP	Number of Annotation parameters	3

Aggregative mapping involving Collection mapping

Component	NoCl	NoCF	LoC	NoXE	NoXA	NoA	NoAP	Total	Result
Java Manual	1	0	64	0	0	0	0	**197**	**1**
Dozer 5.4.0	1	1	9	49	3	0	0	**193**	**2**
Generic DTO Assembler 3.1.0	4	0	72	0	0	12	13	**311**	**1**
Generic DTO Converter 2.0	1	0	8	0	0	4	2	**47**	**5**
jDTO Binder 1.4	1	0	9	0	0	14	16	**122**	**3**
Modelmapper 0.6.1	1	1	26	0	0	0	0	**88**	**4**
Orika 1.4.3	1	0	29	0	0	0	0	**92**	**4**

Point range	Result
< 50	5
50 - 100	4
100 - 150	3
150 - 200	2
>= 197	1

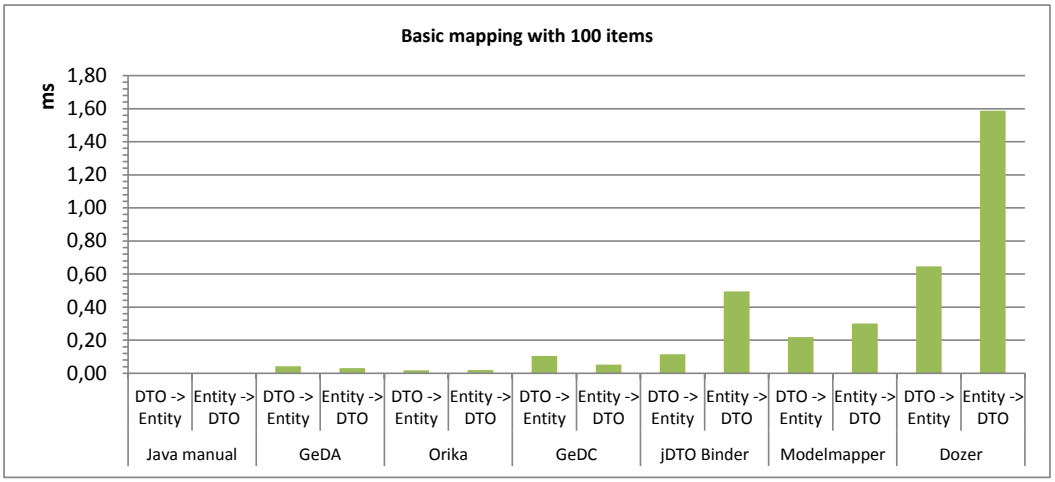
D. APPENDIX: PERFORMANCE TEST RESULTS

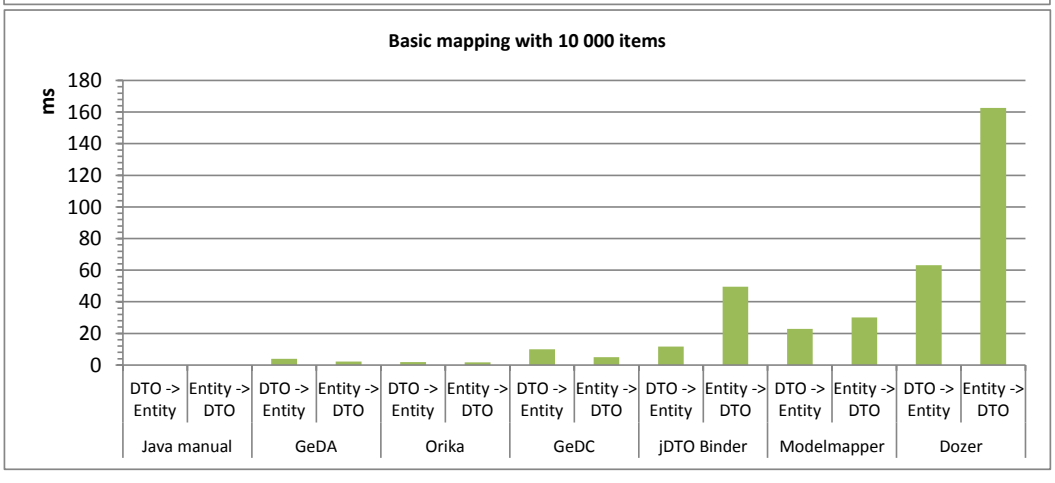
Performance Test Results

Basic aggregative mapping

$T_{manual} : T$ $40*LOG_{10} : LOG_{10}$

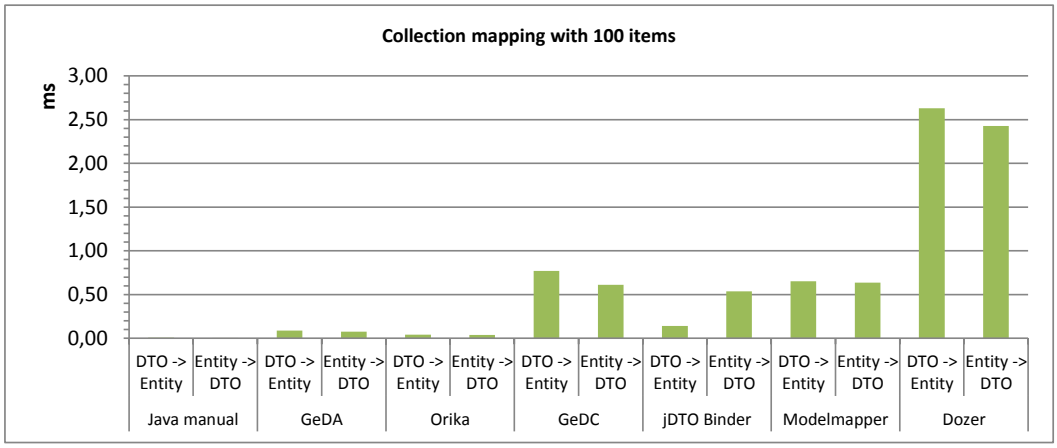
Component	Direction	Time / ns		Scores		Average score
	Items	100	10000			
Java manual	DTO -> Entity	1,29	1,29	**5**	**5**	**5,0**
	Entity -> DTO	2,30	2,3	**5**	**5**	
GeDA	DTO -> Entity	42242,93	3870780,03	**4,8**	**2,8**	**3,9**
	Entity -> DTO	30927,42	2289283,15	**4,9**	**3,1**	
Orika	DTO -> Entity	18532,03	1878510,72	**5**	**3,1**	**4,1**
	Entity -> DTO	18903,73	1768371,64	**5**	**3,2**	
GeDC	DTO -> Entity	104462,19	9934412,86	**4,4**	**2,4**	**3,6**
	Entity -> DTO	51953,57	5023344,86	**4,7**	**2,7**	
jDTO Binder	DTO -> Entity	114677,86	11650884,26	**4,4**	**2,4**	**3,1**
	Entity -> DTO	494518,42	49584505,56	**3,7**	**1,7**	
Modelmapper	DTO -> Entity	218929,43	22872266,27	**4,1**	**2,1**	**3,0**
	Entity -> DTO	301239,57	30141003,97	**3,9**	**1,9**	
Dozer	DTO -> Entity	646422,84	63154293,07	**3,6**	**1,6**	**2,4**
	Entity -> DTO	1588239,72	162661899,63	**3,2**	**1,2**	

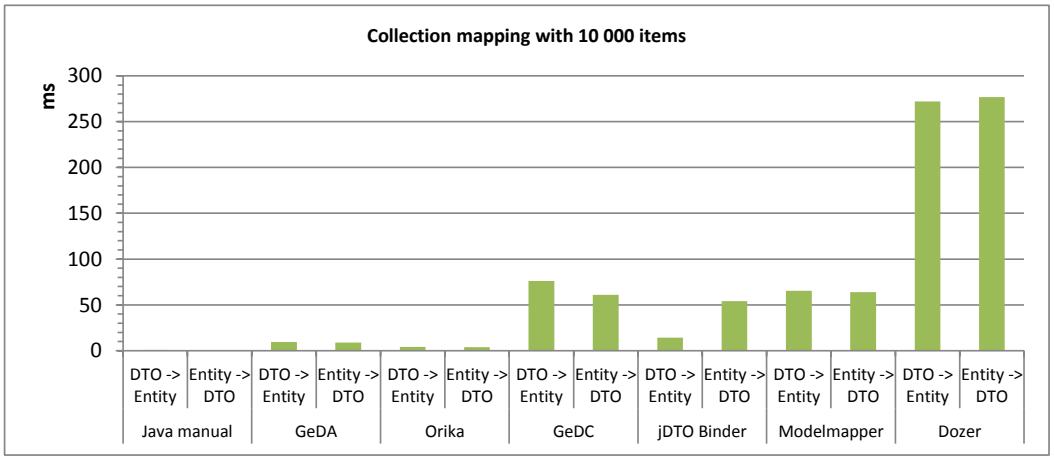




Aggregative mapping involving Collection mapping

Component	Direction	Time / ns		T_{manual} : T	LOG_{10} : LOG_{10}	Average score
	Items	100	10000	Scores		
Java manual	DTO -> Entity	7532,49	754963,03	**5**	**5**	**5,0**
	Entity -> DTO	6051,78	605385,94	**5**	**5**	
GeDA	DTO -> Entity	88847,30	9400684,57	**3,9**	**3,9**	**3,9**
	Entity -> DTO	75243,61	8883397,73	**3,9**	**3,8**	
Orika	DTO -> Entity	40574,99	3980992,04	**4,3**	**4,3**	**4,3**
	Entity -> DTO	39008,13	3745022,94	**4,2**	**4,2**	
GeDC	DTO -> Entity	771221,23	76086776,33	**3**	**3**	**3,0**
	Entity -> DTO	611486,09	61024201,57	**3**	**3**	
jDTO Binder	DTO -> Entity	141601,70	14207696,24	**3,7**	**3,7**	**3,4**
	Entity -> DTO	537849,74	54219229,33	**3,1**	**3**	
Modelmapper	DTO -> Entity	651441,48	65379373,77	**3,1**	**3,1**	**3,1**
	Entity -> DTO	637525,14	63954594,43	**3**	**3**	
Dozer	DTO -> Entity	2628325,41	272014549,3	**2,5**	**2,4**	**2,4**
	Entity -> DTO	2427689,41	276944397,6	**2,4**	**2,3**	





Overall Performance Test Scores

Component	Basic Score	Collection Score	Total
GeDA	3,9	3,9	**4**
Orika	4,1	4,3	**4**
GeDC	3,6	3,0	**3**
jDTO Binder	3,1	3,4	**3**
Modelmapper	3,0	3,1	**3**
Dozer	2,4	2,4	**2**

Test environment	
Java	1.7.0_21 64b Server
Framework	Caliper 0.5-rc1
OS	Windows 8 Pro 64b
CPU	Intel Xeon E3-1230 V2
RAM	16 GB
Date	10th Nov 2013

E. APPENDIX: SCALABILITY TEST RESULTS

Scalability Test Results

Threads	Ramp-up / s	Conversion / request	
1000	20	10000	[Throughput] = req. / min [Medium] = response in ms

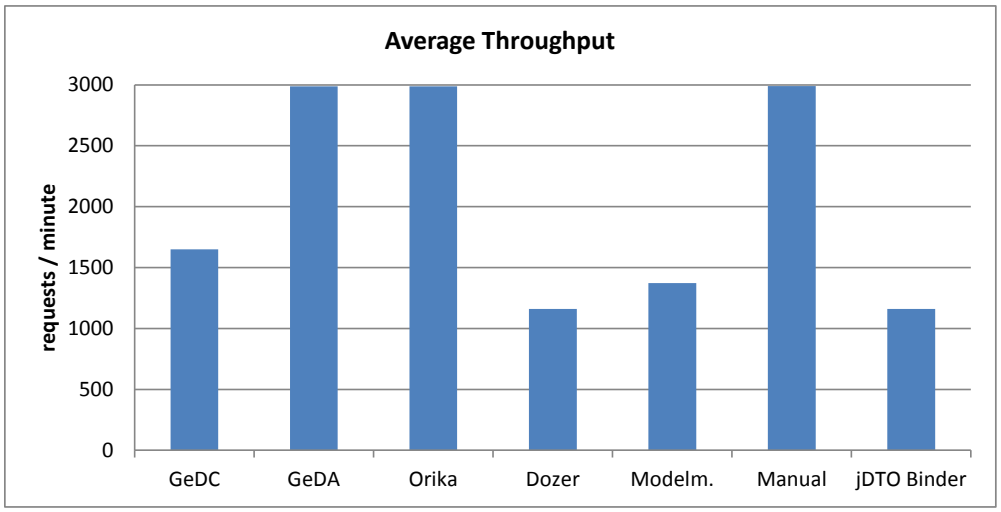
	Round 1		Round 2		Round 3		Average			
Component	**Throughput**	**Medium**	**Throughput**	**Medium**	**Throughput**	**Medium**	**Throughput**	**Medium**	**Errors**	**Result**
GeDC	1608	9058	1660	8948	1680	8769	1649	8925	0	**3**
GeDA	2987	8	2989	7	2989	8	2988	8	0	**5**
Orika	2987	9	2987	9	2986	8	2987	9	0	**5**
Dozer	1151	30000	1172	30000	1161	30000	1161	30000	1	**2**
Modelm.	1258	14946	1467	11405	1390	12893	1372	13081	0	**3**
Manual	2989	1	2990	1	2990	1	2990	1	0	**5**
jDTO Binder	1159	30000	1163	30000	1159	30000	1160	30000	1	**2**

Test environment

Java	1.7.0_21 64b server	**OS**	Windows 8 Pro 64b
Server	Jetty 6.1.0	**CPU**	Intel Xeon E3-1230 V2
JMeter	2.10	**RAM**	16 GB

Implementation Simple Servlets (version 2.5) for each component.
Entity to DTO mapping with initialized mapper component.
One component tested at a time with Jmeter.
Server restarted after each round.

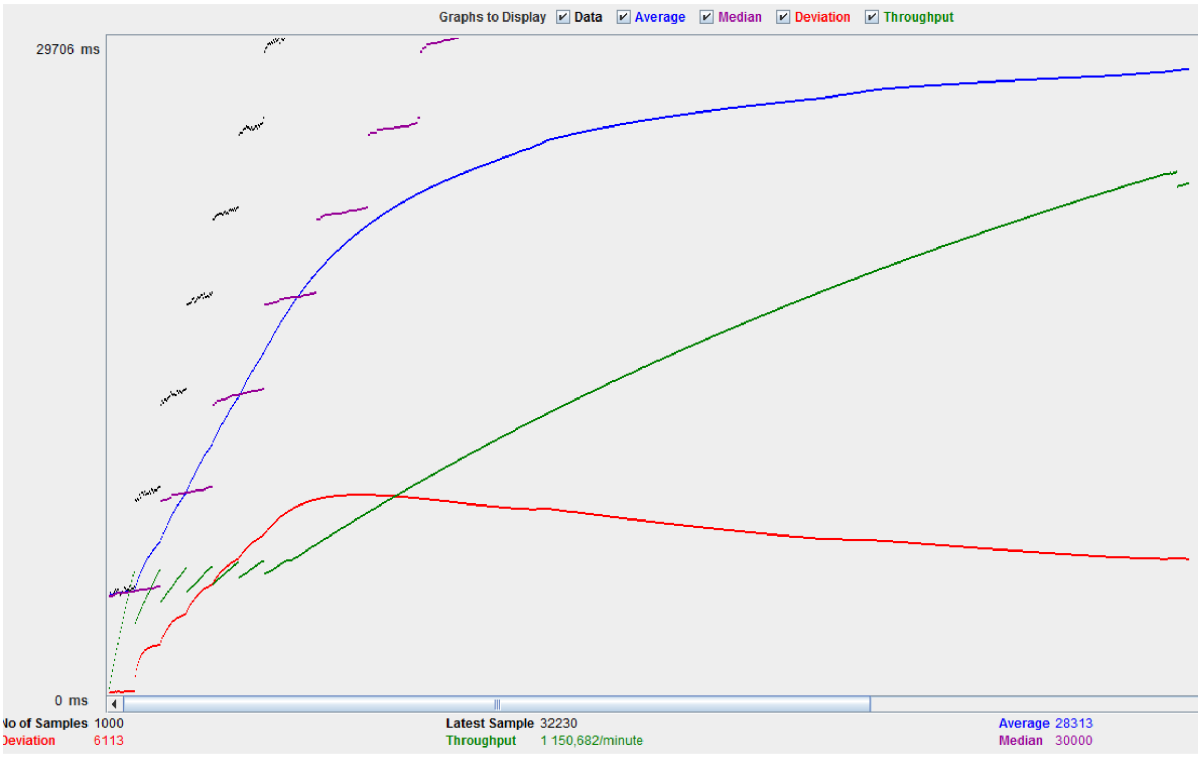
Date 10th Nov 2013



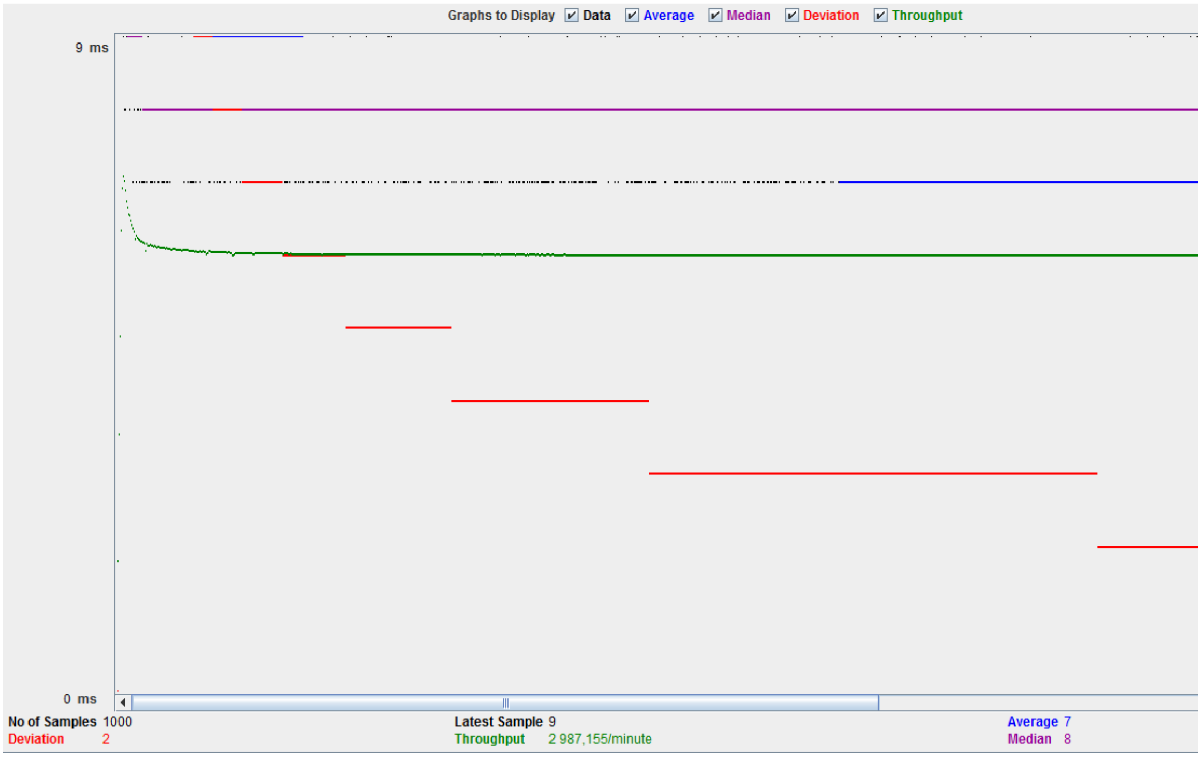
F. APPENDIX: SCALABILITY TEST RESULTS - BEHAVIOR

Scalability Test Results – Behavior

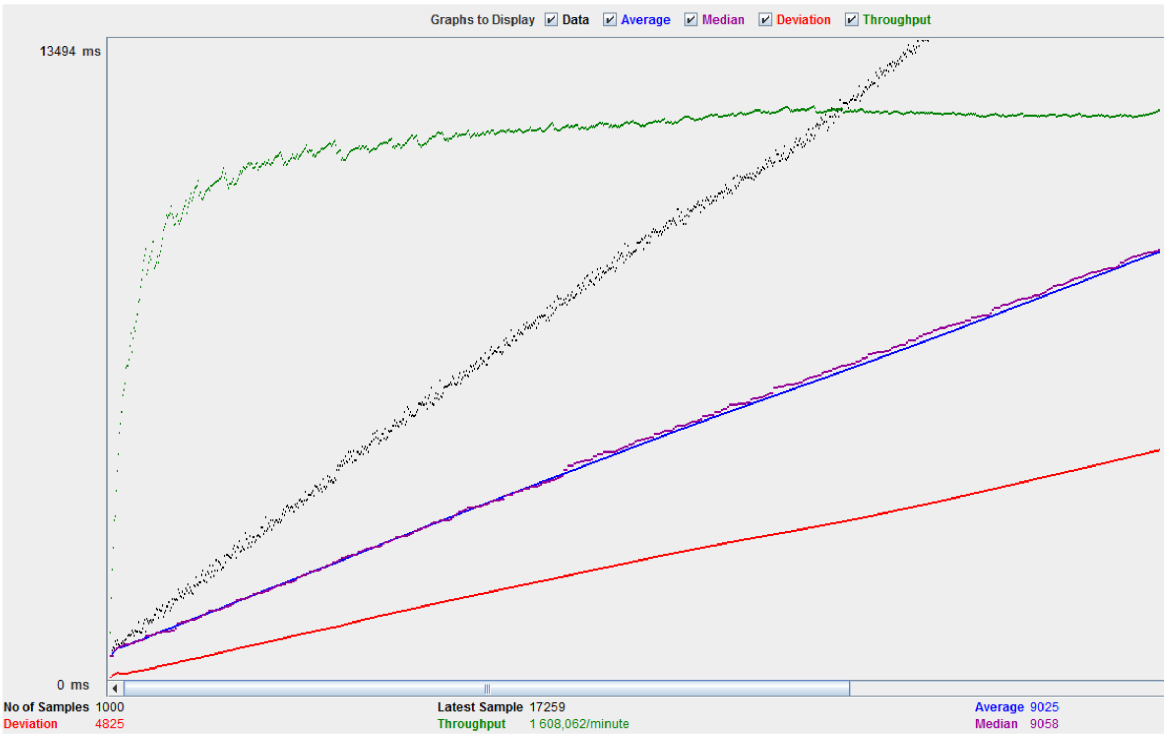
Dozer Round 1:



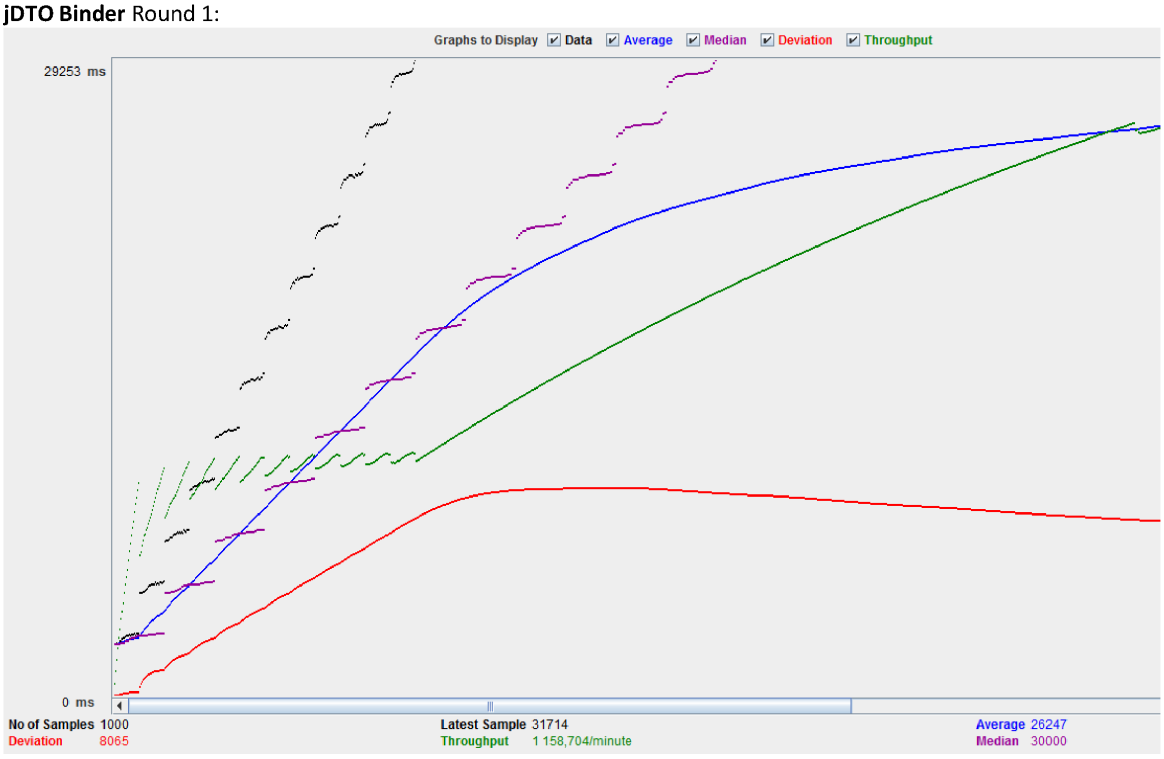
GeDA Round 1:



GeDC Round 1:



jDTO Binder Round 1:



Java Manual Round 1:

Modelmapper Round 1:

Orika Round 1:

G. APPENDIX: EVALUATION CRITERIA FOR OTHER CATEGORIES

Evaluation Criteria for Other Categories

Quality

Meter	Result 5	4	3	2	1
The number of planned releases during the last 12 months	2		1-3		0 or > 3
The number of patch relases during the last 12 months	3-4		1-2 or 5-6		0 or > 6
The number of reported issues during the last 6 months	< 2	2-4	5-10	11-20	> 20
The number of fixed issues compared to reported issues during the last 6 months.	> 75%	60%-75%	45%-60%	25-45%	< 20%
The number of critical bugs opended during the last 6 months.	0	1-2	3-5	5-10	> 10
The average response time to critical bugs during the last 6 months.	< 1 week	1-2 weeks	2-3 weeks	3-4 weeks	> 4 weeks

Support

Meter	Result 5	4	3	2	1
The number of messages posted on project's mailing list or forum during the last six months.	> 100 messages	50-100 messages	16-49 messages	5-15 messages	< 5 messages
The number of StackOverflow questions and answers in total	> 100	50-100	16-49	5-15	< 5

Documentation

Meter	Result 5	4	3	2	1
Existence of various kinds of documentation	Features and extendability documented	All features documented	Basic usage documented	Only test code available	No documentation
User contribution framework	People are allowed to contribute, and contributions filtered by experts		People are allowed to contribute		Users cannot contribute

H. APPENDIX: RESULTS FOR OTHER CATEGORIES

Dozer - Other Categories

Category Quality

Meter	Score	Reasoning
The number of planned releases during the last 12 months	3	Last release 11/112012 -> 1 relase, source: http://dozer.sourceforge.net/releasenotes.html
The number of patch relases during the last 12 months	1	No patch releases.
The number of reported issues during the last 6 months	5	No reported issues in the last 6 months, source: http://sourceforge.net/p/dozer/feature-requests/search/?q=!status%3Awont-fix+%26%26+!status%3Aclosed
The number of fixed issues compared to reported issues during the last 6 months.	5	No bugs, 100%.
The number of critical bugs opended during the last 6 months.	5	No bugs reported, 0.
The average response time to critical bugs during the last 6 months.	5	No bugs reported, 0.
Average:	**4**	

Category Support

Meter	Score	Reasoning
The number of messages posted on project's mailing list or forum during the last six months.	2	8 posts on forum, source: http://sourceforge.net/p/dozer/discussion/452530/
The number of StackOverflow questions and answers in total	5	809 questions, > 100, source: http://stackoverflow.com/search?page=3&tab=relevance&q=dozer
Average:	**4**	

Category Documentation

Meter	Score	Reasoning
Existence of various kinds of documentation	4	All features documented, http://dozer.sourceforge.net/documentation/gettingstarted.html
User contribution framework	3	Documentation included in the project sources, People are allowed to contribute
Average:	**4**	

Generic DTO Assembler - Other Categories

Category Quality

Meter	Score	Reasoning
The number of planned releases during the last 12 months	1	4 releases, http://www.inspire-software.com/confluence/display/GeDA/Revisions
The number of patch relases during the last 12 months	1	0, http://www.inspire-software.com/confluence/display/GeDA/Revisions
The number of reported issues during the last 6 months	3	6 issues, http://www.inspire-software.com/jira/secure/IssueNavigator.jspa?sorter/field=created&sorter/order=DESC and http://www.inspire-software.com/jira/secure/IssueNavigator.jspa?sorter/field=created&sorter/order=DESC
The number of fixed issues compared to reported issues during the last 6 months.	3	3 fixed, 50%
The number of critical bugs opended during the last 6 months.	5	0 critical bugs
The average response time to critical bugs during the last 6 months.	5	No critical bugs
Average:	**3**	

Category Support

Meter	Score	Reasoning
The number of messages posted on project's mailing list or forum during the last six months.	3	24 messages, https://groups.google.com/forum/#!forum/geda-generic-dto-assembler-discussion-group
The number of StackOverflow questions and answers in total	1	1 question and 1 answer, http://stackoverflow.com/search?q=%22generic+dto+assembler%22
Average:	**2**	

Category Documentation

Meter	Score	Reasoning
Existence of various kinds of documentation	4	All features documented
User contribution framework	1	Users cannot contribute
Average:	**3**	

Generic DTO Converter - Other Categories

Category Quality

Meter	Score	Reasoning
The number of planned releases during the last 12 months	3	One release, version 2.0, https://ryh.dy.fi/trac/dtoconverter/wiki/WikiStart?action=history
The number of patch relases during the last 12 months	3	One patch, 1.4.1, https://ryh.dy.fi/mvn2/public/fi/ratamaa/dtoconverter/
The number of reported issues during the last 6 months	2	12 bugs reported, https://ryh.dy.fi/trac/dtoconverter/report/6
The number of fixed issues compared to reported issues during the last 6 months.	5	7 / 9 fixed, 78%, https://ryh.dy.fi/trac/dtoconverter/report/6
The number of critical bugs opended during the last 6 months.	4	1 critical bug, https://ryh.dy.fi/trac/dtoconverter/report/6
The average response time to critical bugs during the last 6 months.	5	< 1 week
Average:	**4**	

Category Support

Meter	Score	Reasoning
The number of messages posted on project's mailing list or forum during the last six months.	1	No mailing list or forum.
The number of StackOverflow questions and answers in total	1	0 questions.
Average:	**1**	

Category Documentation

Meter	Score	Reasoning
Existence of various kinds of documentation	3	Basic usage documented
User contribution framework	1	Users cannot contribute
Average:	**2**	

jDTO Binder - Other Categories

Category Quality

Meter	Score	Reasoning
The number of planned releases during the last 12 months	3	1, version 1.4, https://github.com/jDTOBinder/jDTO-Binder/releases
The number of patch relases during the last 12 months	1	0 patch releases
The number of reported issues during the last 6 months	5	0 bugs, https://github.com/jDTOBinder/jDTO-Binder/issues?state=open
The number of fixed issues compared to reported issues during the last 6 months.	5	No bugs reported.
The number of critical bugs opended during the last 6 months.	5	No bugs reported.
The average response time to critical bugs during the last 6 months.	5	No bugs reported.
Average:	**4**	

Category Support

Meter	Score	Reasoning
The number of messages posted on project's mailing list or forum during the last six months.	1	No forum or mailing list.
The number of StackOverflow questions and answers in total	1	1 question and 1 answer, http://stackoverflow.com/search?q=jdto+binder
Average:	**1**	

Category Documentation

Meter	Score	Reasoning
Existence of various kinds of documentation	4	All features documented
User contribution framework	2	Users are encouraged to participate but no direct channel for this exists.
Average:	**3**	

ModelMapper - Other Categories

Category Quality

Meter	Score	Reasoning
The number of planned releases during the last 12 months	3	3 relases, https://github.com/jhalterman/modelmapper/releases
The number of patch relases during the last 12 months	1	No patch release
The number of reported issues during the last 6 months	1	11+14 = 25 issues, https://github.com/jhalterman/modelmapper/issues, https://github.com/jhalterman/modelmapper/issues?page=1&state=closed
The number of fixed issues compared to reported issues during the last 6 months.	4	14/25 = 56%
The number of critical bugs opended during the last 6 months.	5	No critical bugs
The average response time to critical bugs during the last 6 months.	5	No critical bugs
Average:	**3**	

Category Support

Meter	Score	Reasoning
The number of messages posted on project's mailing list or forum during the last six months.	5	31 threads with total of 134 messages, https://groups.google.com/forum/#!forum/modelmapper
The number of StackOverflow questions and answers in total	2	4 questgions, 3 answers http://stackoverflow.com/search?q=modelmapper
Average:	**4**	

Category Documentation

Meter	Score	Reasoning
Existence of various kinds of documentation	4	All features documented, http://modelmapper.org/user-manual/
User contribution framework	1	Users cannotcontribute
Average:	**3**	

Orika - Other Categories

Category Quality

Meter	Score	Reasoning
The number of planned releases during the last 12 months	3	1 (1.4.0), http://code.google.com/p/orika/wiki/ReleaseNotes
The number of patch relases during the last 12 months	4	3 (1.4.3, 1.4.2, 1.4.1), http://code.google.com/p/orika/wiki/ReleaseNotes
The number of reported issues during the last 6 months	1	33 issues, http://code.google.com/p/orika/issues/list?can=1&q=&sort=-id&colspec=ID+Type+Status+Priority+Milestone+Owner+Summary&cells=tiles
The number of fixed issues compared to reported issues during the last 6 months.	2	11 / 33, 33%
The number of critical bugs opended during the last 6 months.	5	No critical issues
The average response time to critical bugs during the last 6 months.	5	No critical issues
Average:	**3**	

Category Support

Meter	Score	Reasoning
The number of messages posted on project's mailing list or forum during the last six months.	5	147 posts, https://groups.google.com/forum/#!forum/orika-discuss
The number of StackOverflow questions and answers in total	3	9 questions, 7 answers, http://stackoverflow.com/search?q=orika+is%3Aquestion
Average:	**4**	

Category Documentation

Meter	Score	Reasoning
Existence of various kinds of documentation	5	Features and extendability documented, http://orika-mapper.github.io/orika-docs/
User contribution framework	1	Users cannot contribute to documentation
Average:	**3**	

I. APPENDIX: TOTAL SCORES

Evaluation: Dozer

General information

Version: 5.4.0
Lisence: Apache 2.0
Other conditions:

Date: 11.11.2013
Modified: 11.11.2013
Method: Business Readiness Rating, RFC1

Score

Category	Weight	Score	Weight * Score
Functionality	35,0%	4	1,4
Usability	20,0%	2	0,4
Quality	10,0%	4	0,4
Security	0,0%		0
Performance	10,0%	2	0,2
Scalability	10,0%	2	0,2
Architecture	0,0%		0
Support	5,0%	4	0,2
Documentation	10,0%	4	0,4
Acceptance	0,0%		0
Community	0,0%		0
Professionalism	0,0%		0
	100,0%		
		BRR:	**3,2**

Evaluation: Generic DTO Assembler

General information

Version: 3.1.0
Lisence: LGPL
Other conditions:

Date: 11.11.2013
Modified: 11.11.2013
Method: Business Readiness Rating, RFC1

Score

Category	Weight	Score	Weight * Score
Functionality	35,0%	5	1,75
Usability	20,0%	1	0,2
Quality	10,0%	3	0,3
Security	0,0%		0
Performance	10,0%	4	0,4
Scalability	10,0%	5	0,5
Architecture	0,0%		0
Support	5,0%	2	0,1
Documentation	10,0%	3	0,3
Acceptance	0,0%		0
Community	0,0%		0
Professionalism	0,0%		0
	100,0%		
		BRR:	**3,6**

Evaluation: Generic DTO Converter

General information

Version: 2.0
Lisence: MIT
Other conditions:

Date: 11.11.2013
Modified: 11.11.2013
Method: Business Readiness Rating, RFC1

Score

Category	Weight	Score	Weight * Score
Functionality	35,0%	5	1,75
Usability	20,0%	5	1
Quality	10,0%	4	0,4
Security	0,0%		0
Performance	10,0%	3	0,3
Scalability	10,0%	3	0,3
Architecture	0,0%		0
Support	5,0%	1	0,05
Documentation	10,0%	2	0,2
Acceptance	0,0%		0
Community	0,0%		0
Professionalism	0,0%		0
	100,0%		
		BRR:	**4,0**

Evaluation: jDTO Binder

General information

Version: 1.4
Lisence: Apache 2.0
Other conditions:

Date: 11.11.2013
Modified: 11.11.2013
Method: Business Readiness Rating, RFC1

Score

Category	Weight	Score	Weight * Score
Functionality	35,0%	2	0,7
Usability	20,0%	3	0,6
Quality	10,0%	4	0,4
Security	0,0%		0
Performance	10,0%	3	0,3
Scalability	10,0%	2	0,2
Architecture	0,0%		0
Support	5,0%	3	0,15
Documentation	10,0%	3	0,3
Acceptance	0,0%		0
Community	0,0%		0
Professionalism	0,0%		0
	100,0%		
		BRR:	**2,7**

Evaluation: Modelmapper

General information

Version: 0.6.1
Lisence: Apache 2.0
Other conditions:

Date: 11.11.2013
Modified: 11.11.2013
Method: Business Readiness Rating, RFC1

Score

Category	Weight	Score	Weight * Score
Functionality	35,0%	2	0,7
Usability	20,0%	4	0,8
Quality	10,0%	3	0,3
Security	0,0%		0
Performance	10,0%	3	0,3
Scalability	10,0%	3	0,3
Architecture	0,0%		0
Support	5,0%	4	0,2
Documentation	10,0%	3	0,3
Acceptance	0,0%		0
Community	0,0%		0
Professionalism	0,0%		0
	100,0%		
		BRR:	**2,9**

Evaluation: Orika

General information

Version: 1.4.3
Lisence: Apache 2.0
Other conditions:

Date: 11.11.2013
Modified: 11.11.2013
Method: Business Readiness Rating, RFC1

Score

Category	Weight	Score	Weight * Score
Functionality	35,0%	5	1,75
Usability	20,0%	4	0,8
Quality	10,0%	3	0,3
Security	0,0%		0
Performance	10,0%	4	0,4
Scalability	10,0%	5	0,5
Architecture	0,0%		
Support	5,0%	4	0,2
Documentation	10,0%	3	0,3
Acceptance	0,0%		0
Community	0,0%		0
Professionalism	0,0%		0
	100,0%		
		BRR:	**4,3**