



## UNIVERSITY OF PIRAEUS - DEPARTMENT OF INFORMATICS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ – ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

**MSc «.....Informatics.....»**

ΠΜΣ «..... Πληροφορική .....»

### MSc Thesis

#### Μεταπτυχιακή Διατριβή

<b>Thesis Title:</b> Τίτλος Διατριβής:	<b>Benchmarking Software Design patterns in CRM Systems</b> Συγκριτική αξιολόγηση Σχεδιαστικών Προτύπων σε συστήματα CRM
<b>Student's name-surname:</b> Ονοματεπώνυμο φοιτητή:	<b>Emmanouil Prokakis</b> Εμμανουήλ Προκάκης
<b>Father's name:</b> Πατρώνυμο:	<b>Kyriakos</b> Κυριάκος
<b>Student's ID No:</b> Αριθμός Μητρώου:	ΜΠΠΛ2236
<b>Supervisor:</b> Επιβλέπων:	<b>Efthymios Alepis, Professor</b> Ευθύμιος Αλέπης, Καθηγητής

October 2024 / Οκτώβριος 2024

**3-Member Examination Committee**

Τριμελής Εξεταστική Επιτροπή

**Efthymios Alepis**  
**Professor**

Ευθύμιος Αλέπης  
Καθηγητής

**Maria Virvou**  
**Professor**

Μαρία Βίββου  
Καθηγήτρια

**Constantinos Patsakis**  
**Associate Professor**

Κωνσταντίνος Πατσάκης  
Αναπληρωτής Καθηγητής

## Abstract

Software Design Patterns (SDP's) constitute proven solutions to reoccurring problems in software development that provide ready-to-apply solutions for object instantiation, object structure, delegating responsibility and materializing functionality. With the use of Software Quality Metrics (SQMs), we attempt to quantify the quality characteristics of the software, which is constantly evolving. In this present paper, we intend to observe the effects of design pattern introduction in software quality and the reflection of these improvements in SQMs. In a custom CRM Springboot application, which will be built twice (once with design patterns and once without them), we will measure the impact of design patterns in a set of literature - derived metrics. We determined that design patterns have a positive influence in software quality and structure that can be reflected in SQMs, as long as they are applied properly and don't introduce unnecessary complexity.

**Subject Area:** Software engineering

**Keywords:** Software Design Patterns, Software Quality Metrics, Software Quality, Benchmarking, Java, Springboot, CRM

## Abstract in Greek

Τα Σχεδιαστικά πρότυπα αποτελούν έτοιμες και δοκιμασμένες συνταγές για την επίλυση προβλημάτων στη μηχανική λογισμικού. Παρέχουν έτοιμες λύσεις για την αρχικοποίηση αντικειμένων, τη δομή των κλάσεων και τη διαμοίραση των αρμοδιοτήτων. Με τη χρήση μετρικών επιχειρούμε να ποσοτικοποιήσουμε ποιοτικά χαρακτηριστικά μίας υλοποίησης, προκειμένου να αξιολογήσουμε την κατασκευή της. Στο πλαίσιο της παρούσας εργασίας θα δημιουργήσουμε μία CRM εφαρμογή, κατασκευάζοντας το backend σε Java Springboot με και χωρίς τη χρήση των Σχεδιαστικών Προτύπων. Στη συνέχεια, θα αποτιμήσουμε την ποιότητα των δύο υλοποιήσεων με τη χρήση μετρικών που συλλέχθηκαν από τη βιβλιογραφία. Καταλήξαμε στο συμπέρασμα ότι η χρήση των Σχεδιαστικών Προτύπων αποτυπώνεται θετικά στις μετρικές ποιότητας, εφόσον η χρήση τους γίνεται με σωστό τρόπο και δεν εισάγει άσκοπη πολυπλοκότητα.

## **Acknowledgements**

I would like to thank my professor Efthymios Alepis, who, by communicating great knowledge and passion for the subject, always manages to inspire developing engineers to dig deeper into software engineering and evolve their skills.



## Contents

Abstract .....	3
Abstract in Greek.....	3
Acknowledgements .....	4
1. Introductions and Goals.....	6
2.1 Presenting Software Design Patterns.....	6
2.1.1 Introduction .....	6
2.1.2 Benefits of Design Pattern Use .....	7
2.1.3 Design pattern review .....	8
2.2. Discussion on Software Quality Metrics.....	13
3. CRM Application .....	14
3.1 Implementation Architecture .....	15
3.2 Implementation Scope .....	15
3.3 Implementation Analysis .....	25
3.3.1 Client Implementation Presentation.....	25
3.3.2 Design Pattern Implementation Presentation .....	41
3.3.3 Non Design Pattern Implementation Presentation .....	73
4. Benchmarking .....	92
4.1 Establishing a framework for Benchmarking Design Patterns .....	92
4.2 Application to MNS CRM Implementation .....	100
4.3 Results and Observations.....	107
Conclusions.....	108

## 1. Introductions and Goals

In the present paper we intend to benchmark the use of Software Design Patterns using established Software Quality Metrics. The core of a CRM application will be created twice, with one version making use of design patterns and the other doing the opposite. By calculating metrics for both implementations, we hope to see how the introduction of design patterns is reflected on the benchmarking metrics results.

Both CRM back-end implementations will offer the same functionality, but one of them will build certain modules with design patterns included. A collection of metrics will be defined and a custom metric-calculation program will be created to analyze both applications' codebases and calculate the set of metrics for each. We will then focus our efforts on comparing the total / average metrics for both implementations and, following that, we will analyze and compare specific modules - where patterns were introduced - as separate use cases

Our CRM implementations will be written in Java Springboot with the help of JetBrains IDE. We will also deliver a complete front-end user interface for the web-service back-end, that will be built in ReactJS. This is to provide the reader with a complete understanding of the software scope and functionality, so that we can easily get to analyzing the web API code architecture and design pattern use opportunities.

Regarding the paper's structure, in section 2 we will focus on introducing the theory behind design patterns (2.1) and the ideas behind Software Quality Metrics (2.2). In section 3, we will review our CRM implementation's architecture (3.1) and scope (3.2), which will be followed by the complete analysis of our implementations (3.3). Section 3.3.1 will describe and present the front-end ReactJS application, while 3.3.2 and 3.3.3 will focus on the Java back-end codebases with the use and lack of design patterns respectively. Section 4 will include our benchmarking analysis, focusing on establishing a benchmarking framework (4.1), applying the framework on our custom Springboot applications (4.2) and, lastly, we will demonstrate our results (4.3). Section 5 will refer to our conclusions.

## 2.1 Presenting Software Design Patterns

### 2.1.1 Introduction

Software Design Patterns (SDP's) constitute proven solutions to reoccurring problems in Software Development that promote best practices and don't require reinventing the wheel to provide solutions and accomplish certain system behaviors (Mohammed & Elish, 2013; Aversano et al, 2007).

Gamma et al (1995), layed down the first 23 patterns in their book "Design Patterns Elements of Reusable Object-Oriented Software". Said patterns are described as the "Gang of Four patterns" (Ampatzoglou et al, 2013), referencing the four authors of the aforementioned book, who set the stage for the SDP literature. The original 23 patterns were classified into six different groups based on their purpose (Creational, Structural, Behavioral) and scope (class, object).

Creational patterns focus on providing an abstraction for the object instantiation process. Their main goal is to decouple the client code from the creation and composition of objects, while at the same time delivering a simpler interface for use. Using creational patterns, client code can be agnostic to the specific concrete implementation of classes, utilizing only supertype (interface) references to refer to concrete implementations. Structural patterns concentrate on the way

different class hierarchies interact to form more complex structures. According to Aratchige et al (2022), structural patterns help developers form cohesive software architectures by providing best practices for object composition and inheritance. Lastly, Behavioral patterns are interested in providing solutions for delegating responsibilities between different objects and supporting particular actions (Eckel).

All class patterns make use of inheritance to modify a class, compose implementations or provide new behavior, while all object patterns favor object composition to delegate object instantiation or extend with new functionality.

### **2.1.2 Benefits of Design Pattern Use**

The motivation behind establishing specific reusable design patterns is to provide proven solutions that encapsulate best practices and make software implementations more flexible, allowing for reuse of already successful architectures (Gamma et al, 1995). Patterns are meant to make object oriented systems more flexible, maintainable and, certain parts of them, reusable and robust. SDP's entail the essence of know-how in terms of design principles, improving software structure, speeding up delivery and allowing for better communication between developers. Tichy (1998) catalogs the problems solved by applying design patterns as the following: Decoupling, Variant Management, State Handling, Control, Virtual Machines, Convenience patterns, Compound patterns, Concurrency, Distribution. For the scope of this present paper, the effects on decoupling, variant management, state handling and control will become apparent in later sections.

Gamma et al (1995) underline the importance of designing software to be flexible to change and adapt. Software evolution could entail class redefinition and reimplementation that could affect a system in multiple ways and, thus, be costly. Through the use of SDP's, we can help ensure software is allowed to evolve in a certain manner without being forced to refactor large parts of the application. Eckel states that patterns offer a layer of abstraction in order to isolate particular details of an implementation. The goal is to separate parts of the applications that are expected to remain the same from parts that are expected to change, stopping the propagation of changes and general refactoring from materializing.

Design patterns provide templates to design code that avoids many common design flaws. For instance, SDPs promote the use of interface supertype references to objects, which allows decoupling of the client code from the implementation specifics. On the contrary, tight coupling leads to monolithic architectures that require multiple modifications on different classes to support minor changes in functionality. By providing abstractions to the calling environment, patterns are able to limit the knowledge of the client code in regard to class implementation specifics. Another example of how patterns help avoid design flaws is that they show ways of extending functionality without being dependent on inheritance. On more complex systems, subclassing can be difficult to implement since thorough understanding of the parent types is required. Overriding functions, for example, that have mutual dependencies can lead to refactoring and overriding even more methods. SDPs provide more flexibility to extend functionality by applying object composition and delegation of responsibilities to supply new functionality in a more maintainable and easy way.

It is worth mentioning that applying design patterns should always be done to solve a specific problem that is mitigated using the pattern. Applying patterns when not necessary can complicate implementations with unnecessary complexity and overhead.

This paper will focus on the 23 original SDPs, as they were defined in the Gang of Four book. We'll be focusing our attention to a lower layer of abstraction without touching high-level architectural patterns.

## 2.1.3 Design pattern review

### Creational Patterns

The Creational Patterns take up five (5) out of the 23 original Gang of Four patterns. Their main objective is to abstract the object instantiation process by encapsulating it in a separate class. All the calling environment is aware of is just the supertype reference to the instantiated class. Through a polymorphic reference, client-code is separated from the specific implementation details.

#### Singleton (1)

The Singleton design pattern helps enforce that only one instance of a class exists, while ensuring that the entire application has access to the same resource, without the need for global variables (Harmes & Diaz, 2008). Singleton implementation offers a global access point to a class that can only be instantiated once without polluting the namespace.

Controlled access to the sole instance of the target object allows for easy modifications to the functionality or the number of objects that are produced by the Singleton implementation.

#### Abstract Factory (2)

The Abstract factory pattern specializes in producing a set of related products (objects) and is intended for cases where the creation of objects of the same family is needed.

The calling environment is separated from the implementation classes, utilizing only supertype (interface) references to the instantiated objects that result from the a Abstract Factory. Related objects are created in a reusable concrete factory class, that encapsulates the instantiation of the related product family set, while avoiding assumptions of the calling environment in regard to hardcoding implementation classes' names (Gamma et al, 1993; Gamma et al, 1995).

Different product lines can be used by swapping the concrete abstract factory class in order to support a different product family. This also allows for enforcing that the calling environment can only work with a single object family at once.

#### Factory (3)

The Factory pattern delegates the responsibility of instantiating its member objects to subclasses (Harmes & Diaz, 2008; Gamma et al, 1995). Client code will be designed to work with polymorphic references, meaning that it will be possible to work with any implementations of the 'product object' without the need for refactoring.

Performance and efficiency benefits can also become apparent in situations where the creation of objects entails setup operations. These operations can be done once for all objects by the concrete Factory class and, thus, reduce setup costs and duplicate code.

#### Prototype (4)

The Prototype pattern provides the ability to clone existing objects with specific state. Prototype benefits include the simplification of object instantiation, decoupling of the client code from the implementation classes and easier instantiation of complex objects (Gamma et al, 1995).

The cloning of the prototypical instance can also help in mitigating the need to define new subclasses. Cloning ensures that no inheritance will be needed to support different variants of class, where variants are defined based on different state combinations. In such manner, classes will be defined dynamically at runtime and are not statically named during compilation (tight coupling).

#### Builder (5)

As in the previously described patterns, Builder accomplishes the separation of the instance construction process from the calling environment. The Builder pattern is targeted at cases where complex objects need to be built from different parts and the representation of the object should be hidden from the client code.

Essentially, a concrete Builder class offers the functionality to assemble an instance with all the different parts that make it up. Director classes can make calls to the concrete Builder and control the object instantiation process by calling the step-by-step builder functions.

### **Structural patterns**

Structural design patterns focus their attention on object relationships and structure in a way that changes in one part of the system don't require changes elsewhere (Eckel). They heavily rely on object composition in order to achieve more modularity and flexibility by building solid and maintainable architectures. Out of the original 23 patterns, seven (7) are categorized as Structural.

#### Proxy (1)

The Proxy design pattern defines a wrapper class to control access to the target object, also called the real subject (Harmes & Diaz, 2008). It maintains a reference to the target object through object composition, implements the same interface as the real subject and delegates operations to it, while also providing more control over how the target object is consumed by the calling environment (Gamma et al, 1995). Common reasons for using the Proxy pattern are to control the instantiation of resource-expensive objects on demand, enforce variant levels of access on the real subject and ensure thread-safety when accessing the object

#### Decorator (2)

Decorator allows for a flexible extension of functionality of the original object without resorting to inheritance and can be used interchangeably with its target object. This can be done dynamically for specific objects, without having to modify the class hierarchy and statically bind the new behavior (Gamma et al, 1995).

For example, by subclassing we accept that every single instance of a subclass inherits the new behavior of the supertype. On the contrary, through a Decorator / Wrapper class, that implements the same interface and delegates actions to the target object, we have the freedom to choose whether we want a certain behavior to be performed. Thus, new responsibilities can be added to specific objects in a flexible manner.

### Composite (3)

The Composite design pattern makes it so that the client code can treat composite objects (objects that have children objects as part of a hierarchy) the same way as their children objects, which can either be leaves or composite themselves (Harmes & Diaz, 2008). This is achieved through a common interface and provides loose coupling for the individual objects as well as an easy way to traverse the entire hierarchy and perform operations by passing them from one layer to the next. This pattern is suited for when a hierarchy of objects exists and we need to perform an operation to all or a subset of the nodes.

### Adapter (4)

By using the Adapter pattern we can allow two incompatible interfaces to work with one another without modifying the existing implementation (Harmes & Diaz, 2008). The adapter wraps around the class that needs to be adapted and implements the other interface method to achieve compatibility. Essentially, the adapter class translates the method calls of the calling environment into compatible method calls with the existing implementation. The Adapter pattern conforms to the protocol of one class to another (Gamma et al, 1993).

### Bridge (5)

The main goal of the Bridge pattern is to ensure decoupling of the abstraction and the implementation hierarchies, so that they can evolve independently. The abstraction defines the high-level logic and, through object composition, has a reference to the implementor to which it delegates operations. The implementor is more low-level and is related to the specifics of the implementation. Through the Bridge pattern, we can allow for the implementor's implementation to evolve without requiring modifications in the abstraction (higher-level logic). Bridges are very useful because they help make code more modular and improve the flexibility of abstractions.

### Facade (6)

The Facade pattern is dedicated to providing a single, simple and unified interface to client code for accessing a complex subsystem (Gamma et al, 1995). The Facade class acts as an intermediary between client code and the subsystem classes, allowing for decoupling and independent evolution of the subsystem. Subsystem classes can also take advantage of the Facade pattern to layer themselves further. Lastly, the simpler interface exposed to the calling environment also reduces complexity and improves readability of the client code.

### Flyweight (7)

Flyweight aims to help in optimizing an application by sharing state between similar objects when a large number of instances are needed (Harmes & Diaz, 2008). It works by converting a lot of independent objects into a few shared objects for the client code. In order to reduce the number of instances, state is segregated into intrinsic state (data shared with the objects inside Flyweight class) and extrinsic state (data that is unique to the calling environment and can't be shared). In addition, since the objects are shared and to avoid problems with creating instances, the Flyweight pattern also employs the use of a factory to handle instance creation (Gamma et al, 1995).

## Behavioral Patterns

According to Eckel, behavioral patterns specialize in handling certain types of actions, like algorithms, iteration, separation of responsibilities or fulfilling a request. Gamma et al (1993) note that behavioral patterns have to do with how classes interact with each other and how they distribute operations. They define strategies for class cooperation and communication to materialize a specific functionality, since tasks cannot be completed by just one class alone. Out of the original 23 Gang of Four patterns, 11 are in the behavioral subset.

### Chain of Responsibility (1)

The core idea of the Chain of Responsibility (CoR) pattern is to decouple the sender and the receiver of a message by allowing the message to travel through a “chain” of different handlers (Gamma et al, 1995). This becomes especially useful if we don't know ahead of time which handler is supposed to take care of a specific request. Handlers are chained together by utilizing object composition, in a structure that resembles a standard linked list implementation. In case the request is not meant to be handled by a certain handler, it can be passed down to the next one.

When making use of the CoR pattern, both the sender and the receiver of the message have no knowledge of each other. Responsibilities can be separated on different handlers and the chain can be modified dynamically on runtime, allowing for a lot of modularity and flexibility for the implementation.

### Observer (2)

The Observer pattern defines a one-to-many relationship between a subject and its observers, ensuring that any changes in the subject will be disseminated to its observers (Gamma et al, 1995). This pattern is prevalent in event-handling systems and promotes loose coupling between the subject and the observers. All the subject knows is the interface of observer items and their implementation can be defined dynamically at runtime without the need for any assumptions by the subject. The communication between the subject and its dependent observers adheres to the pub-sub principle.

### Iterator (3)

By using the Iterator pattern we enable traversing a collection object (iterable) without exposing its underlying structure to the calling environment (Gamma et al, 1995). The Iterator encapsulates the logic for accessing and traversing the list object, allowing for different traversal implementations, decoupling from the client code (since there is no need to accommodate the internal representation of the list) and simplification of the latter. Different collections can be traversed by using the Iterator pattern due to its support for polymorphic iteration through a common interface for iterable objects.

### Command (4)

According to Harmes & Diaz (2008), the Command pattern enables the parameterization of a method call along with the decoupling of the object invoking the action from its implementation. All Command objects share a common interface that defines an execute method to encapsulate a request.

By employing the Command pattern, we achieve better control over the handling of requests, opening the way for queues, undo / redo functionality, logging / reapplying changes and modeling transaction centric systems (Gamma et al, 1995). Additionally, new commands can be added easily without the need for modifications of the existing implementation. This is because the above-mentioned pattern promotes the decoupling of the abstraction from its representation.

#### Strategy (5)

The Strategy pattern encapsulates different algorithms / behaviors in classes defined by a common interface, permitting interchanging the strategy (i.e. an encapsulated algorithm) used by a context class. Object composition is used to delegate tasks to the strategy (Gamma et al, 1995). Due to the common interface, the actual implementation is defined dynamically at runtime while ensuring the decoupling of the context class and the actual algorithm that gets executed. Strategy works as a better replacement of inheritance for implementing different behaviors for the same operation / method. For instance, defining three different behaviors for a class means hard-coding the implementation on each subtype. With the Strategy pattern, we can decouple the implementation logic from the context, making it more modular and flexible.

#### Mediator (6)

With the Mediator pattern we can control multiple interactions between a group of objects while the Mediator class acts as an intermediary between them (Gamma et al, 1995). A concrete Mediator class holds references to the colleague objects (object composition) and promotes decoupling by handling all their interactions. Thus, the colleague objects never directly reference each other, reducing the number of interconnections in the system and, as a consequence, the need for refactoring when changes are due. The Mediator class provides a central point of control for handling object interactions, abstracting the interaction process and shifting focus away from the individual behaviors of colleagues, which are encapsulated inside their respective classes.

#### Interpreter (7)

The Interpreter pattern specializes in modeling the grammar of a simple language making it possible to represent and interpret sentences (Gamma et al, 1995). Different classes are used for representing grammar rules and contribute to building abstract syntax trees to support the modeling of the language rules and grammar.

#### Visitor (8)

By using the Visitor pattern, new functionality can be added to existing objects without having to modify their classes (Gamma et al, 1995). New operations can be encapsulated in new Visitor classes promoting the Separation of Concerns and the reduction of class pollution with additional methods. Since Visitor classes contain the entire logic of the new operation, they provide a central point for handling the new functionality and reusing it when needed. Objects that consume Visitor classes will accept a Visitor and then call that Visitor's method by passing themselves as context.

#### Memento (9)

While respecting class encapsulation, the Memento pattern allows for an object's internal state to be saved in a snapshot Memento object and restored later (Gamma et al, 1995). The aforementioned pattern can act as the backbone of an undo-redo mechanism (with the additional support of a stack-like data structure) or helping a system recover from errors. Encapsulation is protected because only the "originator" object is able to create instances of the Memento and restore back to them.



## Template Method (10)

The Template Method pattern grants us the ability to specify the core structure of an algorithm while at the same time deferring to subclasses the implementation of certain algorithm steps (Gamma et al, 1995). The subtypes can't redefine the skeleton of the algorithm, but are only capable of defining the implementation of specific steps (methods); the structure of the algorithm is maintained. Through the non-overridable supertype method (which defines the algorithm skeleton), the parent class makes calls to the method implementations of the subclasses.

## State (11)

The State pattern is great for cases where changes in the object's internal state should be reflected on the operations performed by the target object (Gamma et al, 1995). Large conditional logic structures can be avoided since the State pattern makes it so that the operation performed will be defined at runtime, based on the target's internal state.

Each possible condition is modelled in its own State class, promoting the Single Responsibility principle and the Separation of Concerns. Encapsulating a specific operation in a separate State class offers great flexibility (adding a new operation requires no modification in the existing implementation) and moves the system away from multipart monolithic conditional statements.

## 2.2. Discussion on Software Quality Metrics

Software Quality Assurance (SQA) refers to a process of monitoring software engineering methods to ensure quality in the final product (Lee, 2014). Software Quality Metrics (SQM's) are utilized by SQA to quantify the specifics of software quality. According to Trivedi & Kumar (2012), they attempt to quantify the quality and performance characteristics during the build phase of the software. The latter is under the process of Software Evolution, during which modifications and enhancements take place. Due to complexity, SQM's play a vital role in keeping Software Quality under control during Software Evolution (Drouin, Badri & Toure, 2013).

Different SQM's have been conceived through the years in an attempt to quantitatively express the quality characteristics that define "well written" software.

Still used old metrics include Lines of Code (LOC), number of functions and lines of comments (Molnar, 2020). When referring to these old metrics, Singh et al (2011) argue that counting modules are a better way to estimate the size of a piece of software than functions, with module being a part of the application that can be compiled independently. In the case of LOC, comments and blank lines can be excluded. Size could also be estimated by counting the number of tokens in the codebase. In 1977, Halstead proposed, among others, the Halstead Program Volume (HV) metric to assess the size, and therefore complexity, of a program (Lee, 2014; Singh et al, 2011).

Moving away from quantification of size, McCabe (1976) proposed the idea of Cyclomatic Complexity (CC) in Software, applying Graph Theory to Software Engineering to address the control flow of the application. The CC / Cyclomatic Number, coming from Graph Theory, refers to the number of independent paths through a codebase (Singh et al, 2011). Lee (2014) states that CC counts the number of decision elements in a program, adding the number of conditions, the number of decisions and the value of 1.

An attempt by Henry & Kafura (1984) was made to measure Information Flow Complexity (IFC), referring to the volume of information going in and out of a system (Lee, 2014). IFC takes into account the size of software (quantified by lines of code, number of functions/modules) along with

information flows into and out of the system, underlying the importance of the complexity added due to integrations with other components (Lee, 2014; Singh et al, 2011).

The Maintainability Index (MI) was introduced in the early 1990s by Oman and Hagemeister (1992) as a metric for quantifying software maintainability. MI is a composite metric that utilizes older metrics, such as, HV, CC and LOC, with applied weights (Welker, 2001). It focuses on the magnitude of operations and operands, logic complexity and the size of the codebase. The calculation formula is the following:

$$MI = 171 - 5.2 \times \log_2(HV) - 0.23 \times CC - 16.2 \times \log_2(LOC)$$

Chidamber and Kemerer (1994) introduced the CK SQM suite targeting object oriented systems, which includes six different metrics (Drouin, Badri & Toure, 2013).

- Weighted Methods per Class (WMC) counts the cyclomatic complexity of each method of class and sums it up in a per class metric.
- Number of Children (NOC) is a per class metric that sums the number of direct subtypes of a class, indicating which classes are being heavily reused for inheritance. In general, even though a lot of inheritance signifies reusability, having a large amount of subclasses in the same hierarchy level can be considered a “bad” characteristic that can be mitigated by the use of design patterns (e.g. Decorator, Strategy).
- Coupling Between Objects (CBO) refers to the number of coupled classes of a class which is valid when access to another classes method and members takes place. Tight coupling between modules can appear problematic in terms of maintenance and flexibility, due to having to make modifications in one part of a system because of changes elsewhere. Most design patterns incorporate some type of supertype interface reference to promote loose coupling of classes.
- Depth of the Inheritance Tree (DIT) focuses on the longest path from the root supertype to its target subtype. A longer path could mean higher complexity due to the total amount of inherited properties and hard-wired static behavior.
- Response For a Class (RFC) is interested in a set of methods in a class along with the method calls in code outside of the boundaries of said class. Essentially, it counts the number of methods that can be executed in response to a message of an object of the class.
- Lack of Coherence in Methods (LCOM) indicates the dissimilarity between methods based on the attributes accessed by those methods. It's a per class metric of cohesion with the intention of establishing if a class adheres to the Single Responsibility principle. Methods that are not related to each other should be separated into different classes.

### 3. CRM Application

In the scope of this present paper, effort will be put into creating a basic CRM web application. The purpose of this implementation is to build the CRM back-end logic twice, once with and once without the use of design patterns. Our main goal is to observe the improvements SDPs are expected to achieve in regards to software quality, flexibility, maintainability and reliability.

The CRM Application will be implemented with ReactJS for the front-end and Java Springboot in the back-end. We want the single React application to interact with each of the backend implementations interchangeably, since the design pattern analysis will be performed on the Java backend. Even though the benefits of a microservices architecture are clear (O'Connor, Elger & Clarke, 2017; De Laurentis, 2019), we will proceed with a monolithic design of the backend logic in order to maximize the interaction between different components and make the use of design patterns more essential. More specifically, the effects of tight coupling will be more prevalent in a monolithic system, due to the number of class interconnections and interdependencies.

Some inspiration on CRM functionality, referring completely to what a CRM does (not design or implementation), was taken from the Open Source SuiteCRM application. The [official website](#) was utilized to retrieve information about CRM entities (e.g. Accounts, Contacts, Leads etc) and processes (Lead Conversion). The application created in the scope of this paper is for educational use only and is in no way an alternative / competitor to a complete CRM solution like SuiteCRM or any other.

The CRM application of this paper will be named MNS CRM.

### **3.1 Implementation Architecture**

The CRM logic will be implemented in Java and the Springboot framework. The CRM Application UI will be created with React for the front-end side of things. React will perform REST API calls to the back-end and will send / receive JSON requests. We want the UI element of the CRM App to be separate from the back-end logic for many reasons, but, in the scope of this paper, it is important to be able to use each back-end implementation interchangeably with the same React client.

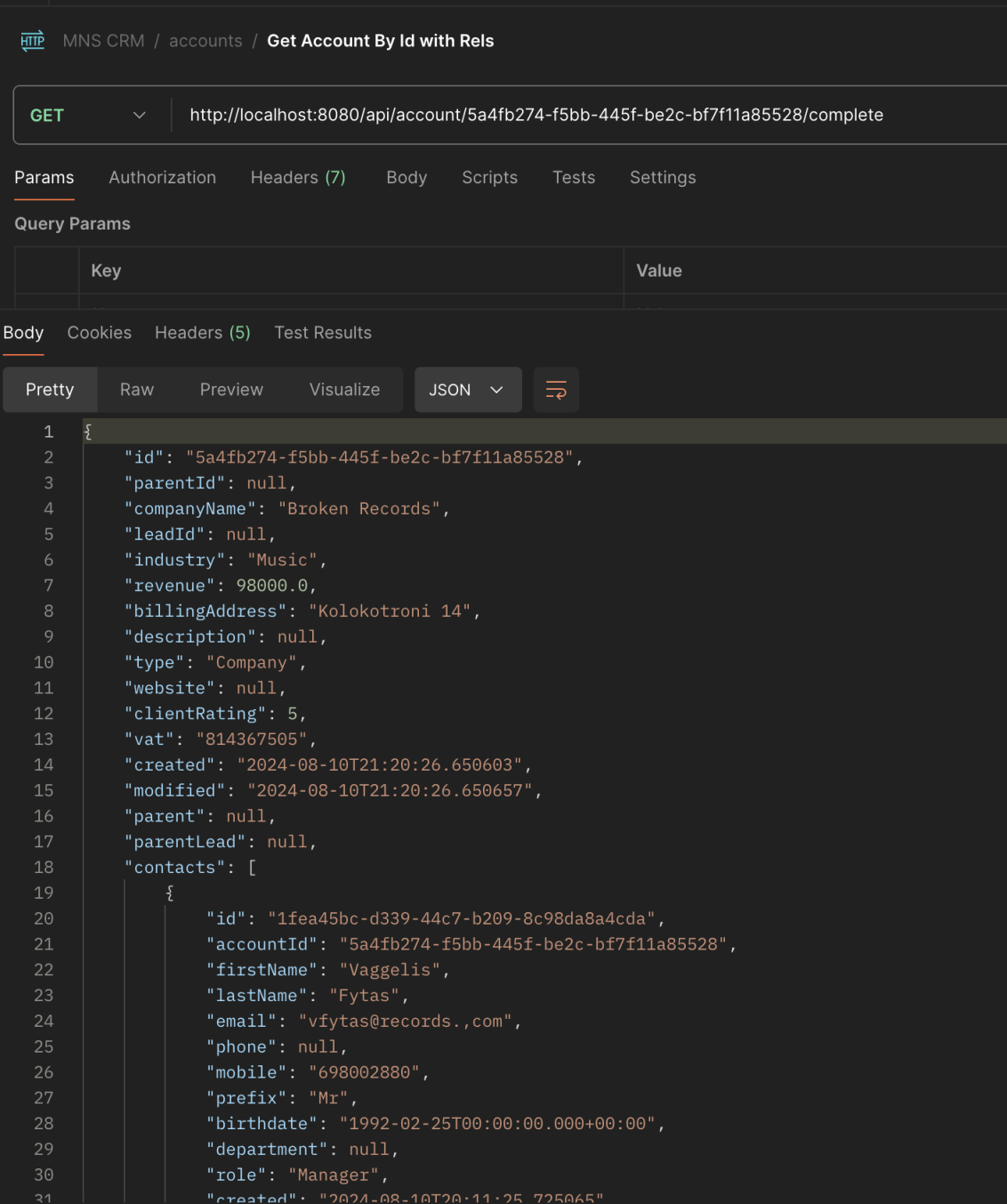
### **3.2 Implementation Scope**

MNS CRM will support core CRM tables, henceforth called "Entities", like Accounts (end customers), Contacts (customer staff), Leads (potential customers), Opportunities (sales attempts), Tasks, Cases (support tickets and inquiries) and Voice Calls (phone communications with end customers). Database records will be called Entity entries. We will present the database schema in visual diagrams that were made with the help of dbdiagram.io.

For every Entity, there should be the ability to perform CRUD operations through the API endpoints. Each entry will have its own page, where the user will be able to view, edit, delete and create new entries. Lists will also be available for all Entities to allow for a tabular view of the entries. The rendering of the Entity attributes in the Entity view & edit pages will be done dynamically in the front-end, based on the JSON response provided by the Springboot web service.

Accounts will be the core Entity of the CRM. They represent end-customer entities, with whom the users of the CRM interact. Accounts have a self-reference (one-to-many), allowing for a hierarchical structure to be maintained. Additionally, Accounts have a one-to-many relationship with Contacts, Opportunities, Cases and VoiceCalls. An Account can also optionally have a one-to-one relationship with a Lead entry.





The screenshot displays a REST client interface for a GET request. The URL is `http://localhost:8080/api/account/5a4fb274-f5bb-445f-be2c-bf7f11a85528/complete`. The response is shown in JSON format, detailing an account entry with various attributes and a list of associated contacts.

```
1  {
2    "id": "5a4fb274-f5bb-445f-be2c-bf7f11a85528",
3    "parentId": null,
4    "companyName": "Broken Records",
5    "leadId": null,
6    "industry": "Music",
7    "revenue": 98000.0,
8    "billingAddress": "Kolokotroni 14",
9    "description": null,
10   "type": "Company",
11   "website": null,
12   "clientRating": 5,
13   "vat": "814367505",
14   "created": "2024-08-10T21:20:26.650603",
15   "modified": "2024-08-10T21:20:26.650657",
16   "parent": null,
17   "parentLead": null,
18   "contacts": [
19     {
20       "id": "1fea45bc-d339-44c7-b209-8c98da8a4cda",
21       "accountId": "5a4fb274-f5bb-445f-be2c-bf7f11a85528",
22       "firstName": "Vaggelis",
23       "lastName": "Fytas",
24       "email": "vfytas@records.,com",
25       "phone": null,
26       "mobile": "698002880",
27       "prefix": "Mr",
28       "birthdate": "1992-02-25T00:00:00.000+00:00",
29       "department": null,
30       "role": "Manager",
31       "created": "2024-08-10T20:11:25.725065"
```

Figure 1.2. Retrieving an Account entry.

The screenshot shows a REST client interface for a GET request to the endpoint `http://localhost:8080/api/contact/1fea45bc-d339-44c7-b209-8c98da8a4cda/complete`. The response is displayed in JSON format, showing a contact object with a parent company object.

```

1  {
2    "id": "1fea45bc-d339-44c7-b209-8c98da8a4cda",
3    "accountId": "5a4fb274-f5bb-445f-be2c-bf7f11a85528",
4    "firstName": "Vaggelis",
5    "lastName": "Fytas",
6    "email": "vfytas@records.,com",
7    "phone": null,
8    "mobile": "698002880",
9    "prefix": "Mr",
10   "birthdate": "1992-02-25T00:00:00.000+00:00",
11   "department": null,
12   "role": "Manager",
13   "created": "2024-08-10T20:11:25.725065",
14   "modified": "2024-08-10T21:26:15.91987",
15   "parent": {
16     "id": "5a4fb274-f5bb-445f-be2c-bf7f11a85528",
17     "parentId": null,
18     "companyName": "Broken Records",
19     "leadId": null,
20     "industry": "Music",
21     "revenue": 98000.0,
22     "billingAddress": "Kolokotroni 14",
23     "description": null,
24     "type": "Company",
25     "website": null,
26     "clientRating": 5,
27     "vat": "814367505",
28     "created": "2024-08-10T21:20:26.650603",
29     "modified": "2024-08-10T21:20:26.650657",

```

**Figure 1.3. Retrieving the child Contact.**

Contacts model staff that is employed in an Account entry, thus there is a one-to-many relationship between Accounts and Contacts. Contacts can also have a one-to-many relationship with Cases. This is because when a Case is opened for an Account, communication takes place between the CRM user and an employee of the end-customer.

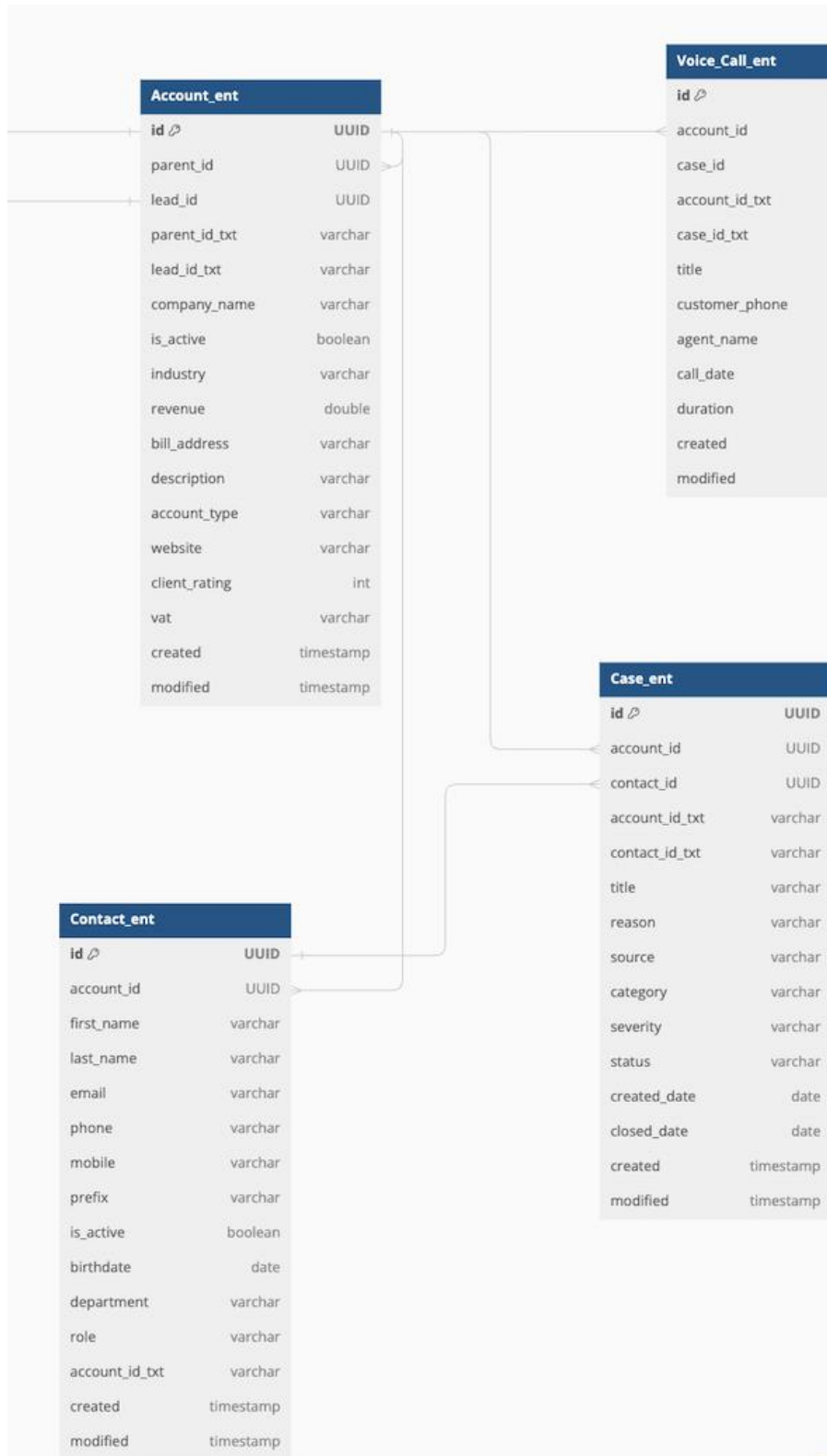


Figure 1.4. Visual Representation of Contact data-model and relationships

Leads refer to potential customers that can go through the Sales funnel. They have no parent entries and, once converted, will be translated into an Account, a Contact and an Opportunity

entry. Leads maintain a one-to-many relationship with Tasks, since specific task-actions might be needed to successfully convert a Lead.

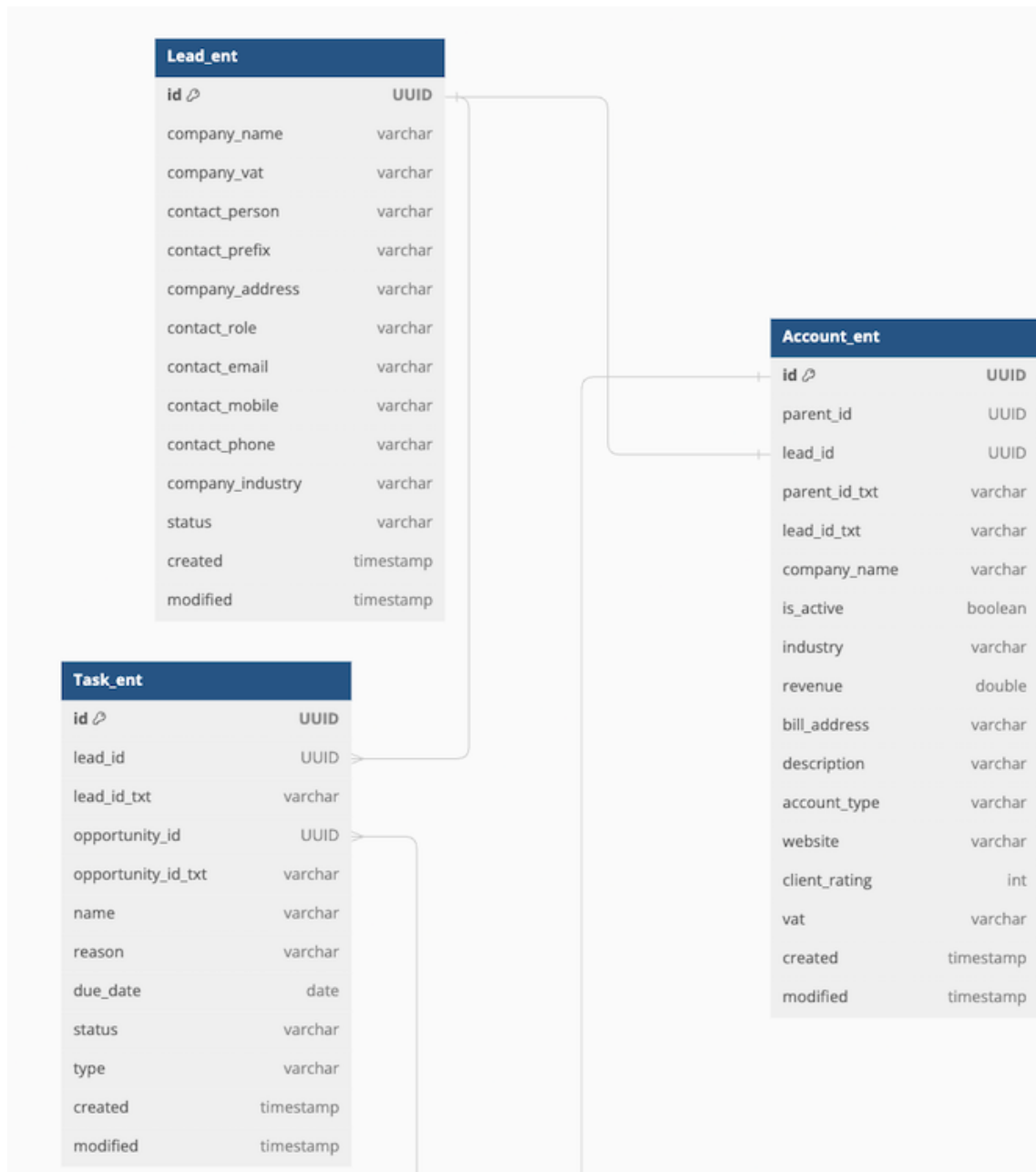


Figure 1.5. Visual Representation of Lead data-model and relationships

Opportunities are the actual sales opportunities where, if successful, a sale is made and invoiced. Opportunities are always related to a parent Account and have their own Status lifecycle. There is a one-to-many relationship between Opportunities and Tasks, because certain task-actions might be needed to guide an Opportunity to success.



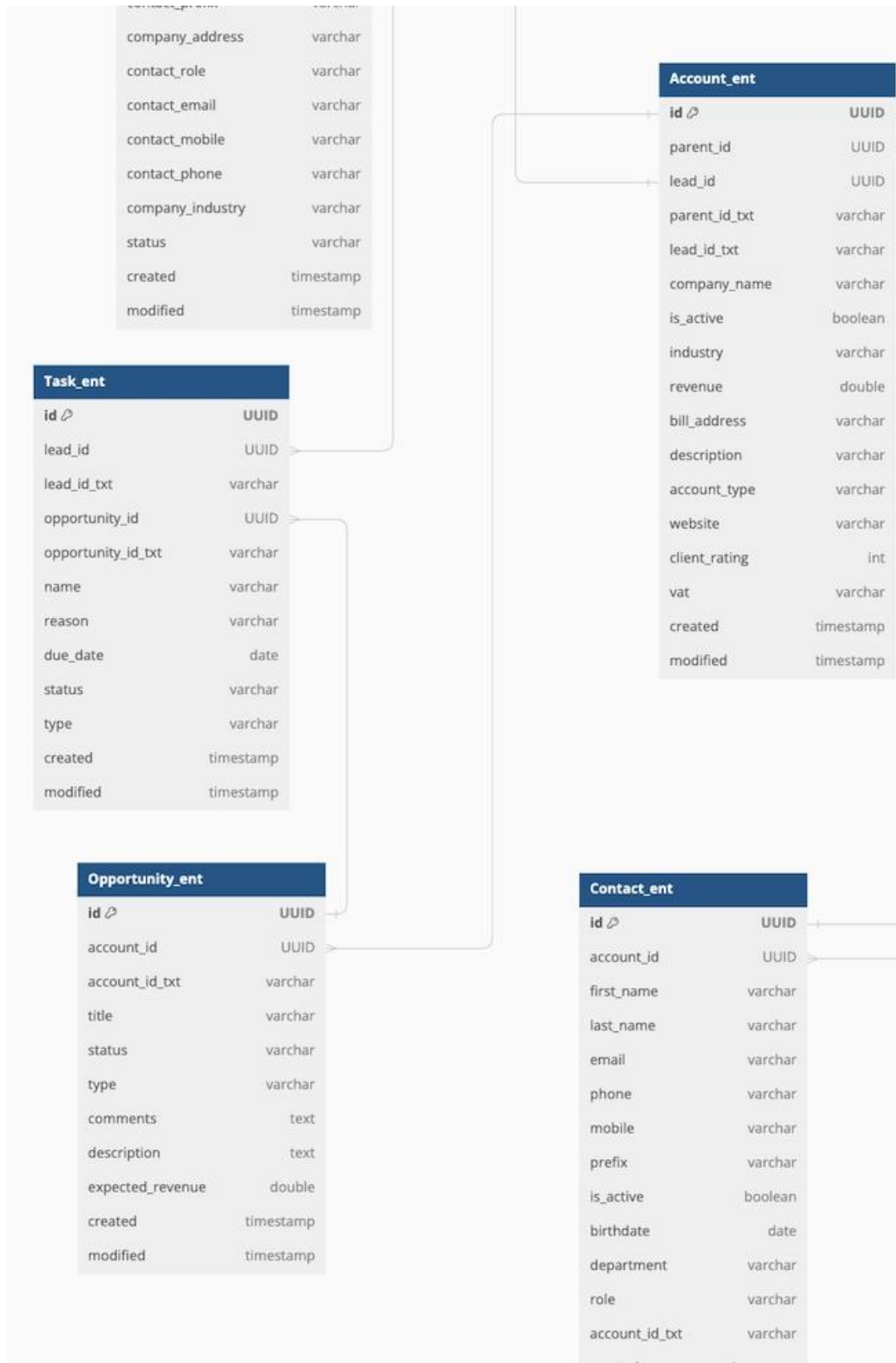


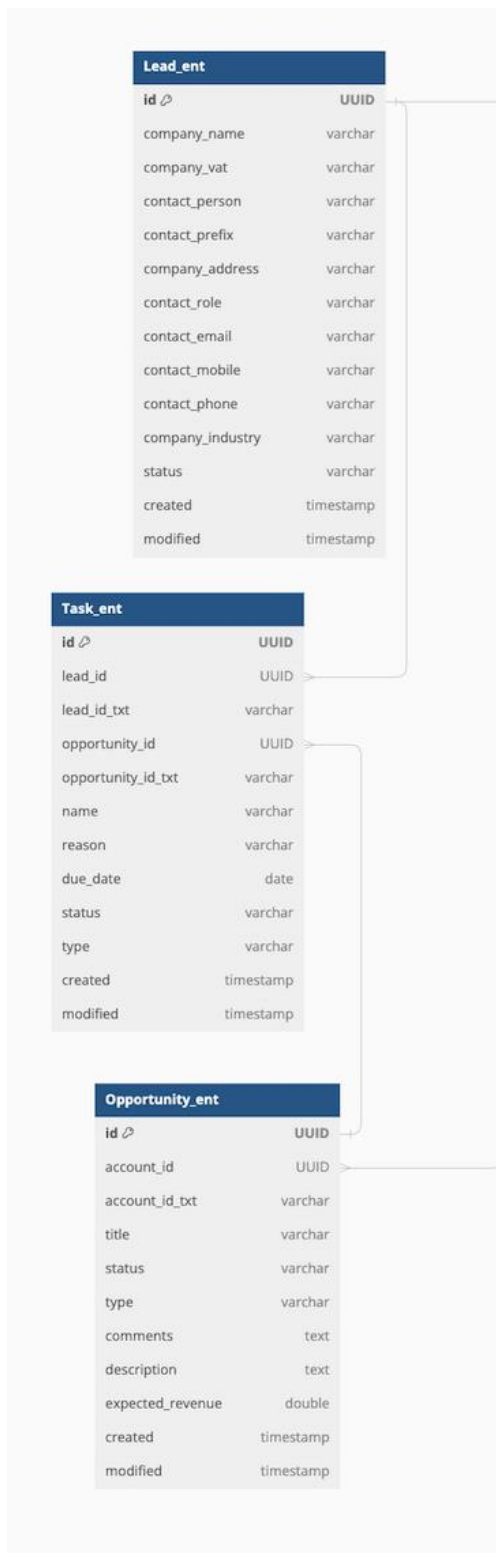
Figure 1.6. Visual Representation of Opportunity data-model and relationships

Cases belong in the “support” Entities of MNS CRM and are used for logging, handling and solving Customer Support related issues for Accounts. There is a parent reference for an Account and a Contact entry and Cases have a one-to-many relationship with VoiceCalls, since multiple communications with the client might be needed to resolve the Case.



Figure 1.7. Visual Representation of Case data-model and relationships

Tasks model specific actions that need to be completed before a certain date. They have no children references and always point to a parent entry that can be either a Lead or an Opportunity.



**Figure 1.8. Visual Representation of Task data-model and relationships**

VoiceCalls are also included in the “support” Entities of the present implementation. For every phone-call / voice communication with the end-customer, a VoiceCall entry is maintained in MNS

CRM. VoiceCalls are always related to an Account (many-to-one) and a Case entry (many-to-one).

This is the complete database schema of MNS CRM.

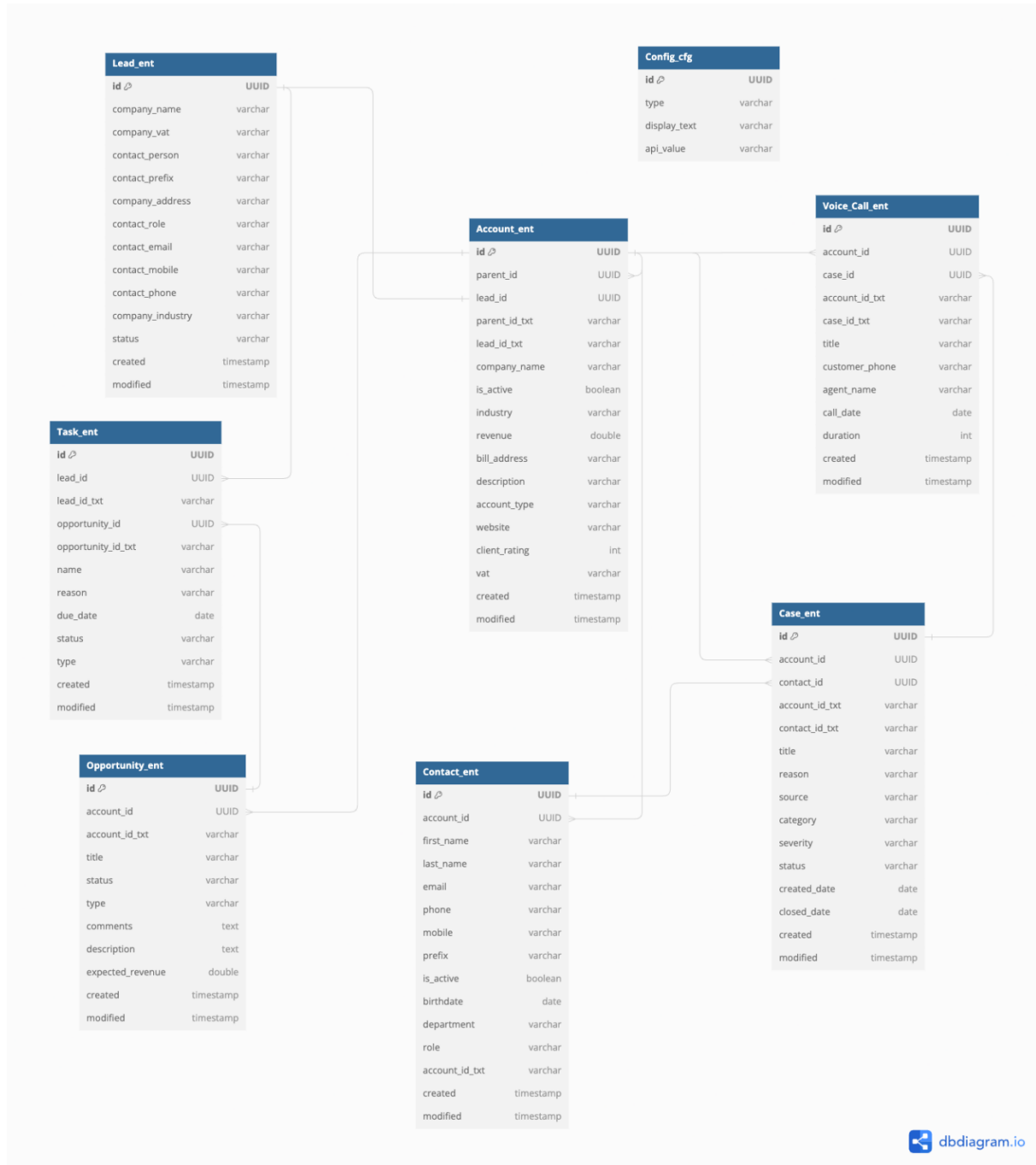


Figure 1.9. Complete Database schema, made with dbdiagram.io

MNS CRM will support two core sales process, the Lead Conversion and the Discount Calculation process. During a Lead Conversion, a potential customer is led through the Sales funnel. If the handling is successful, a Lead can be converted to an Account, a Contact and an Opportunity entry, citing the source Lead’s data. The Discount process will consider the end-customer’s industry, revenue and loyalty to provide a discount on an amount.

## 3.3 Implementation Analysis

### 3.3.1 Client Implementation Presentation

Before taking a look at the implemented solution in the back-end, we believe it would be beneficial to review the application front-end to get a better understanding of the functionality as end users of the CRM. The entire source code for the ReactJS application can be found at [https://github.com/emmprokac/mns\\_crm\\_client](https://github.com/emmprokac/mns_crm_client).

When the web app loads, users are placed in the “Overview” page, that includes information about the CRM.

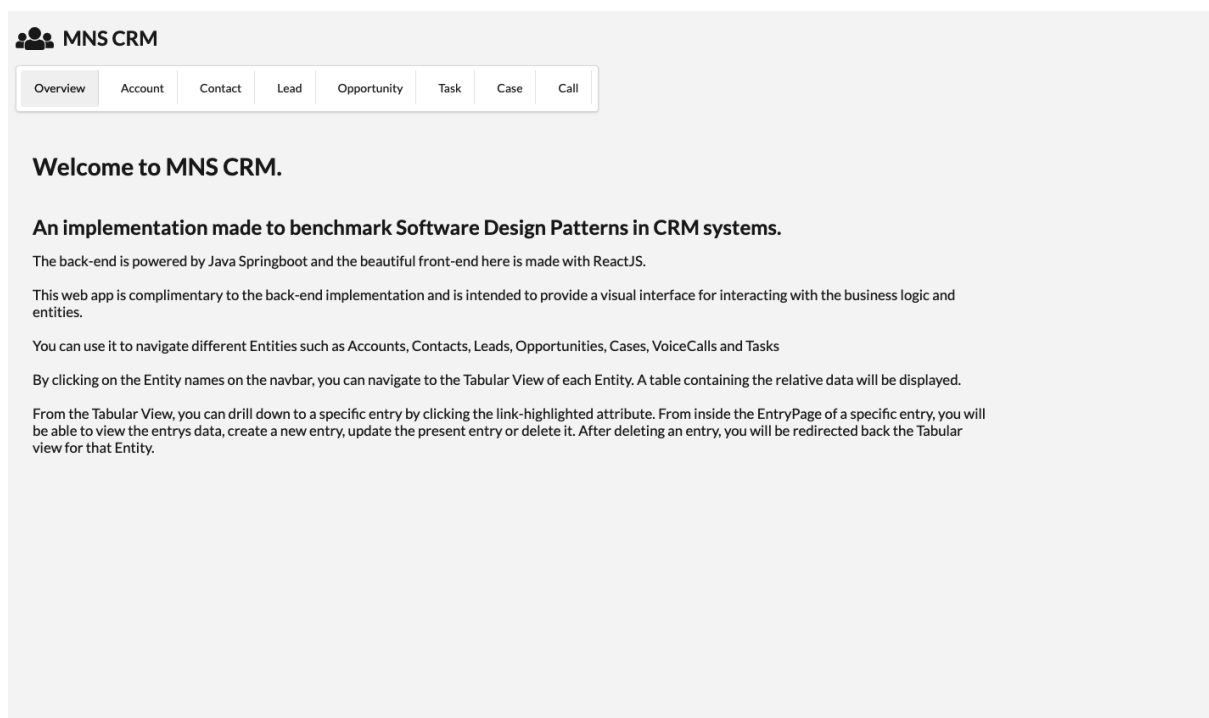


Figure 2.1. Client App landing page

By clicking on the Entity names placed on the navbar, the user can navigate to the TabularView of multiple Entities. For each Entity, a data-table that provides a “tabular” representation of database records is rendered.

**MNS CRM**

Overview Account Contact Lead Opportunity Task Case Call

**Viewing Account entries**

Company Name	Industry	Yearly Revenue	Billing Address	Description	Type	Website	Client Rating	VAT	Created On	Last Updated On	Parent	Source Lead	Is Active
Estate Menidi Group	Real Estate	2000000	Dekelias 35	a great real estate company	Company	esmenidi.gr	10	798002855	2024-08-10 20:11	2024-08-10 20:14	-	-	yes
Testopoulos Inc	Insurance	25000	Testopoulou 55	the first thing when insurance comes to mind	Company	testopoulos.gr	6	896640720	2024-08-10 20:11	2024-08-10 20:15	-	-	yes
Artopoulos AE	Food	35000	Dimitriou 82	bread company	Person	psomi-tora.gr	8	799570819	2024-08-10 20:11	2024-08-10 20:16	-	-	yes
Menidi Housing	Housing	10000	Sevastou 11	housing for all	Company	-	4	952117684	2024-08-10 20:18	2024-08-10 20:18	Estate Menidi Group	-	yes
Broken Records	Music	98000	Kolokotroni 14	-	Company	-	5	814367505	2024-08-10 21:20	2024-08-10 21:20	-	-	yes

Figure 2.2. Tabular View of Account entries

**MNS CRM**

Overview Account Contact Lead Opportunity Task Case Call

**Viewing Contact entries**

First Name	Last Name	Email	Phone	Mobile	Prefix	Birthdate	Department	Role	Created On	Last Updated On	Parent	Is Active
George	Kanellopoulos	gkanel@artopoulos.com	-	6973200494	Mr	2000-08-31	-	Manager	2024-08-10 20:11	2024-08-10 21:21	Artopoulos AE	yes
Vaggelis	Fytas	vfyas@records.com	-	698002880	Mr	1992-02-25	-	Manager	2024-08-10 20:11	2024-08-10 21:26	Broken Records	yes
Kostas	Mimikopoulos	kmimiko@menidi.gr	-	6920009221	Mr	-	Communication	Employee	2024-08-10 21:22	2024-08-10 21:22	Menidi Housing	yes
Giannis	Mimikopoulos	gmimiko@menidi.gr	2102722887	6983244898	Mr	-	Director	CEO	2024-08-10 21:23	2024-08-10 21:23	Menidi Housing	yes
Xristos	Georgopoulos	xgeo@menidi.gr	2108871324	694532001	Mr	-	-	CEO	2024-08-10 21:24	2024-08-10 21:24	Estate Menidi Group	yes
Kostantinos	Riskakis	krisk@records.com	-	6932141919	Mr	-	-	Employee	2024-08-10 21:28	2024-08-10 21:28	Broken Records	yes

Figure 2.3. Tabular View of Contact entries

**MNS CRM**

Overview Account Contact Lead Opportunity Task Case Call

**Viewing Opportunity entries**

Title	Status	Type	Description	Comments	Expected Revenue	Created On	Last Updated On	Related Account
Sell PC's	New	upsell	new ddr-5 tech	client has repeatedly asked for	2200	2024-08-10 20:11	2024-08-10 21:31	Broken Records
Provide Consulting	New	upsell	help with tech migration	-	1000	2024-08-10 21:32	2024-08-10 21:32	Menidi Housing
Sell MNS Server Software	Quote	sell	-	-	3000	2024-08-10 21:33	2024-08-10 21:33	Artopoulos AE

Figure 2.4. Tabular View of Opportunity entries

The implementation of the Tabular View is fully generic and dynamic for all Entities. The API request that retrieves the entry list is made by the App component and the entryList is then passed down to the TabularView component. The first record entry is fetched and parsed to identify the data-table columns. A loop inside the JSX section, renders the columns dynamically. A second loop follows, which is responsible for dynamically rendering a Row for every record and a Cell for every field of every record. This implementation provides great flexibility since it easily establishes a working TabularView for new Entities that might be added in the future.

```

<div>
  <Header as="h2">Viewing {entityName} entries </Header>

  {
    recordList?.length > 0 ?
    <Table celled color="red">
      <TableHeader>
        <TableRow>
          {fields.map(field => (
            <TableHeaderCell key={"h" + field}>{LabelMapper.fieldNameToLabelMap[field]}</TableHeaderCell>
          ))}
        </TableRow>
      </TableHeader>

      <TableBody>

        {recordList.map(row => (
          <Table.Row key={"tRow" + row.id}>

            {fields.map(col => (
              <Table.Cell>
                collapsing={false}
                textAlign="center"
                verticalAlign="middle"
                key={"rec" + row["id"] + col}

                <TabularCell recordObject={row} fieldName={col} cellClicked={fieldClicked} entityName={entityName}/>
              </Table.Cell>
            ))}

          </Table.Row>
        ))}

      </TableBody>
    </Table>
  }

```

Figure 2.5. Tabular View generic implementation

In case no entries are found for an Entity, an informative message appears with button-triggered modal for creating a new entry.

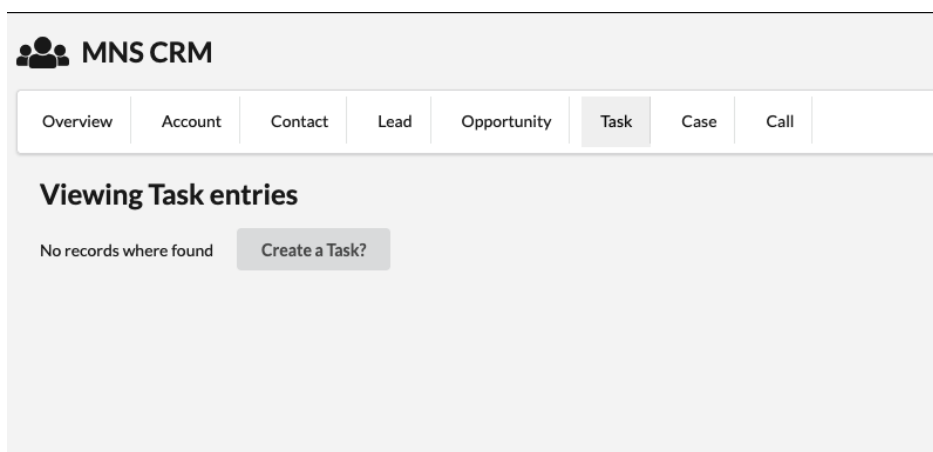


Figure 2.6. No records found page for Tasks

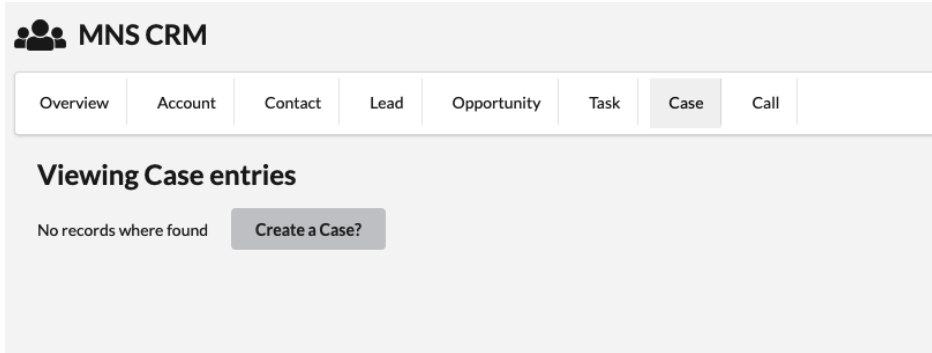


Figure 2.7. No records found page for Cases

```

<div>
  <div>
    <span>No records where found</span>
    <Button onClick={modalButtonPressed} style={{marginLeft: "20px"}}>Create a {entityName}?</Button>
  </div>
  <div>
    <TabularModal entityName={entityName}
      childModalClosed={modalClosed} showModal={noRecordModalOpen} modalKey={modalKey}/>
  </div>
</div>
}
    
```

Figure 2.8. No records founds page implementation

By clicking a the linkable-attribute of an Entity entry, the user will be navigated to the Entry page for that record.

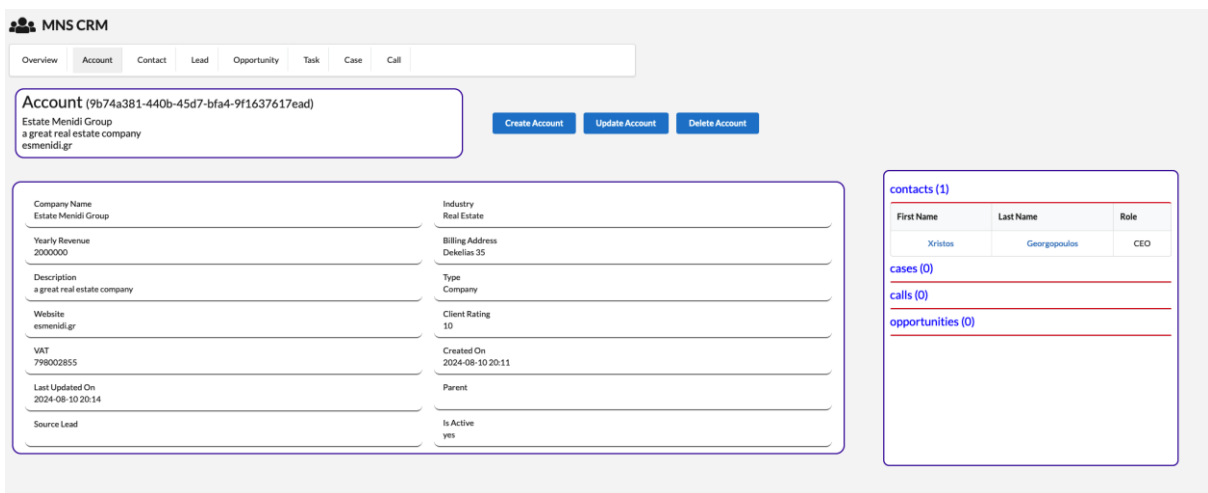


Figure 2.9. Account Entry Page example



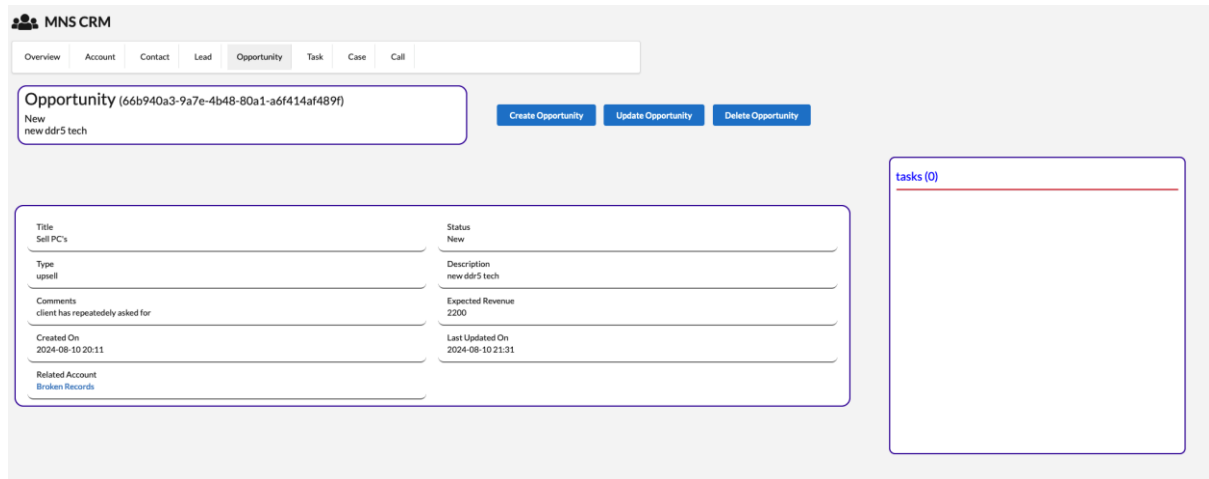


Figure 2.10. Opportunity Entry Page example

The entry page can be divided into four (4) sections. The top-left and top-right components refer to the Entry Header and the Entry actions. The former includes the Entity name, the Entity id and a couple of important attributes for each Entity. Through the viewableFields list, the rendering of the Entry Header is dynamic for all Entities. The same is true for the latter, where every action-button is generically created for the relative Entity entry.

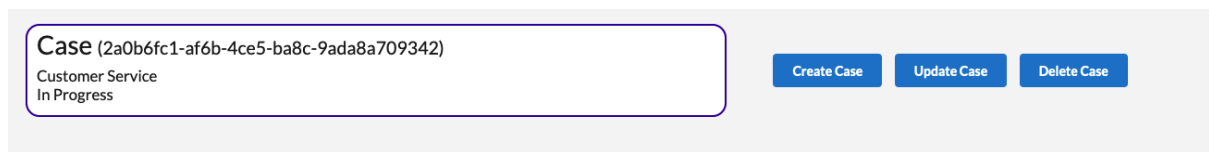


Figure 2.11. Entity Header example for Cases



Figure 2.12. Entity Header example for Accounts

```

<div className="header-top-container">
  <div className="entity-header">
    <div className="entity-header-object">
      {Parse.firstLetterCapital(entityName)} <span style={{fontSize: "0.7em"}}>{record["id"]}</span>
    </div>

    <div className="entity-header-preview-fields">
      {
        fieldCollection.map(field => (
          viewableFields.includes(field) ?
            <div key={"head-" + field} style={{marginRight: "20px"}}>
              <span key={"head-" + field}>{record[field]}</span>
              <br/>
            </div>
          :
            <span key={"head-" + field}></span>
        ))
      }
    </div>
  </div>

  <div className="actions-container">
    {
      standardActions.map(act => (
        <EntityAction key={act} entityName={entityName} entryId={record["id"]} actionLabel={act} actionClicked={actionClicked}/>
      ))
    }
    {
      renderLeadConversionAction()
    }
  </div>
</div>

```

Figure 2.13. Rendering of Entity Header components implementation

The bottom-left component of the Entity page is referred to as the main data container and includes the attribute values of the relative entry.

First Name Kostas	Last Name Mimikopoulos
Email kmimiko@menidi.gr	Phone
Mobile 6920009221	Prefix Mr
Birthdate	Department Communication
Role Employee	Created On 2024-08-10 21:22
Last Updated On 2024-08-10 21:22	Parent <a href="#">Menidi Housing</a>
Is Active yes	

Figure 2.14. Main data area example of Contact

Title Sell PC's	Status New
Type upsell	Description new ddr5 tech
Comments client has repeatedly asked for	Expected Revenue 2200
Created On 2024-08-10 20:11	Last Updated On 2024-08-10 21:31
Related Account <a href="#">Broken Records</a>	

Figure 2.15. Main data area example of Opportunity

Following the paradigm set by other parts of the implementation, the rendering of these fields is also dynamic, defined by the fields included in the JSON response from the web service. The response object's attributes are separated into two columns and each field is rendered dynamically in one of the two columns.

```
function separateFieldsIntoTwoGroups(){
  if(!entry){
    return;
  }

  let idx = 0;
  const leftList = [];
  const rightList = [];
  const total = [];

  const relFields = [];

  for(let field of Object.keys(entry)){
    if(nonRenderableFields.includes(field)){
      continue;
    }

    // related entry arrays
    if(Array.isArray(entry[field])){
      Logger.log("added to rel fields");

      Logger.log(field);
      relFields.push(field);
      continue;
    }

    if(idx % 2 === 0){
      leftList.push(field);
    }else{
      rightList.push(field);
    }

    total.push(field);
    idx++;
  }

  setFirstColFields(leftList)
  setSecondColFields(rightList);
  setFieldTotal(total);
  setRelationshipFields(relFields);
  Logger.log(relFields);
}
```

Figure 2.16. Dynamic field parsing logic

```

<div className="main-record-data-container">
  <div className="field-group-container">
    <div className="left-div">
      {
        firstColFields.map(field => (
          <EntityPageField key={field} fieldName={field} fieldValue={entry[field]} entityName={entityName}
            relatedEntrySelected={relatedRecordSelected}/>
        ))
      }
    </div>

    <div className="right-div">
      {
        secondColFields.map(field => (
          <EntityPageField key={field} fieldName={field} fieldValue={entry[field]} entityName={entityName}
            relatedEntrySelected={relatedRecordSelected}/>
        ))
      }
    </div>
  </div>
</div>

```

Figure 2.17. Dynamic rendering of fields in two column

```

function renderFieldValue(){
  if(typeof fieldValue === 'object'){
    const [id, displayValue] = Parse.handleObjectValue(fieldValue, fieldName, entityName)
    const relatedObjectType = getRelatedObjectType(fieldValue);
    return <a onClick={entryObjectSelected} data-id={id} data-object={relatedObjectType}
      style={{cursor : "pointer"}}>{displayValue}</a>;
  }

  return Parse.parseTableValue(fieldValue, fieldName);
}

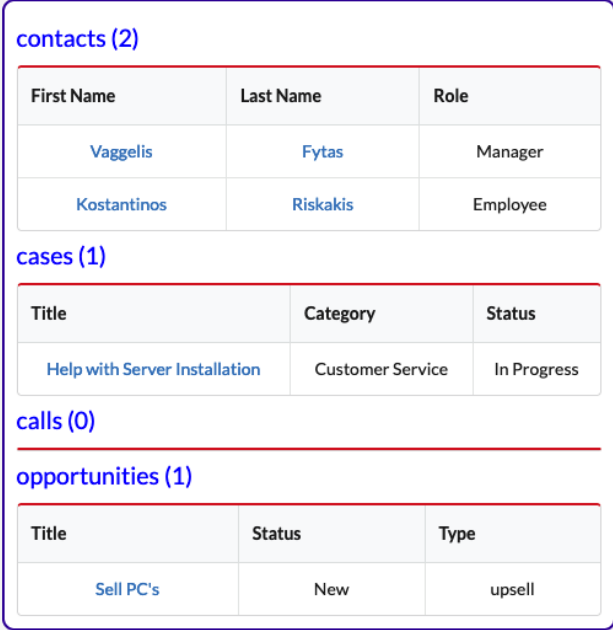
function entryObjectSelected(event){
  relatedEntrySelected(EventGenerator.getRelatedLinkEvent(event.target.dataset.id, event.target.dataset.object))
}

return(
  <div className="field-container">
    <label htmlFor={fieldName} className='field-label'>{LabelMapper.fieldNameToLabelMap[fieldName]}</label>
    <br/>
    <div>
      <span className='field-value'>
        {renderFieldValue()}
      </span>
    </div>
  </div>
)

```

Figure 2.18. Implementation of EntityPageField component

All related fields are also collected in a separate list variable for the fourth and last component of the Entry page. The related records area is responsible for rendering mini-tabular views for the children entries of the displayed Entity entry.



**contacts (2)**

First Name	Last Name	Role
Vaggelis	Fytas	Manager
Kostantinos	Riskakis	Employee

**cases (1)**

Title	Category	Status
Help with Server Installation	Customer Service	In Progress

**calls (0)**

**opportunities (1)**

Title	Status	Type
Sell PC's	New	upsell

**Figure 2.19. Related records area example for Account**



**cases (1)**

Title	Category	Status
Help with Server Installation	Customer Service	In Progress

**Figure 2.20. Related records area example for Contact**

The rendering of each children list is dynamically handled by the RelatedEntriesGroup component which, in turn, loads the RelatedEntriesTable component.

```

{
  relationshipFields?.length > 0 ?
    <div className="related-records-area">
      {
        relationshipFields.map(field => (
          <RelatedEntriesGroup key={"rel" + field} entityName={entityName} relationshipName={field}
            entryId={entryId} relatedEntriesList={entry[field]}
            relatedEntrySelected={relatedRecordSelected} />
        ))
      }
    </div>
    :
    <div></div>
}

```

Figure 2.21. Rendering each related list dynamically implementation

```

return (
  <div>
    <div style={{margin: "10px 0"}}>
      <span style={{color: "blue", fontSize: "1.4em"}}> {relationshipName} ({relatedEntriesList?.length})</span>
    </div>
    <div>
      <RelatedEntriesTable relationshipName={relationshipName} originEntityName={entityName}
        recordList={relatedEntriesList} relatedRecordClicked={relatedEntrySelected}/>
    </div>
  </div>
)

```

Figure 2.22. RelatedEntriesGroup component implementation

All clickable fields will navigate the user on the entry that the linkable attribute represents. This is possible through a generated event, which is bubbled up across the entire component hierarchy. All parts of the implementation that support these links make use of the EventGenerator class, which provides a centralized generator of events to be handled by the EntryPage and App components.

```

class EventGenerator{
  static getEvent(name, source, value){
    return {
      name, source, value
    }
  }
  static getRelatedLinkEvent(entryId, entityName){
    return {
      entryId, entityName
    }
  }
}

```

Figure 2.23. EventGenerator class implementation

Users can also interact with the action buttons. They provide access to “create”, “update” and “delete” functionality and clicking an action button causes a modal to appear. Create modals contain no data and are responsible for inserting a new entry to the database. Update modals automatically load the currently present entry of the Entry page and every modification on these data will be interpreted as an update to the currently present entry. The delete modal contains no

data and will handle the deletion of the currently present entry from the database. The modal action buttons are colour-coded, so that the user has one more sign about the operation he/she is about to commit.

## Create Contact

**Add values**

<b>First Name</b> First name	<b>Last Name</b> Last name
<b>Email</b> Email	<b>Phone</b> Phone
<b>Is Active</b> <input type="checkbox"/> Active	<b>Mobile</b> Mobile
<b>Parent Account</b> Select Account	<b>Prefix</b> None
<b>Birthdate</b> XXXX-XX-XX	<b>Department</b> Department
<b>Role</b> None	

**Figure 2.24. Create Contact Modal example**

## Update Contact

**Add values**

First Name <input type="text" value="Kostantinos"/>	Last Name <input type="text" value="Riskakis"/>
Email <input type="text" value="krisk@records.com"/>	Phone <input type="text" value="Phone"/>
Is Active <input checked="" type="checkbox"/> Active	Mobile <input type="text" value="6932141919"/>
Parent Account <input type="text" value="Broken Records"/>	Prefix <input type="text" value="Mr"/>
Birthdate <input type="text" value="XXXX-XX-XX"/>	Department <input type="text" value="Department"/>
Role <input type="text" value="Employee"/>	

Figure 2.25. Update Contact Modal example

## Delete Contact

Are you sure you want to delete Contact with id: 9c54ca8c-ac42-4eec-a16e-555b90dc19ef

Figure 2.26. Delete Contact Modal example

If a Lead's Status is set to "Success", a "Convert" action will also appear. That operation will trigger the Lead Conversion process for that Lead entry. This modal button is also colour-coded.



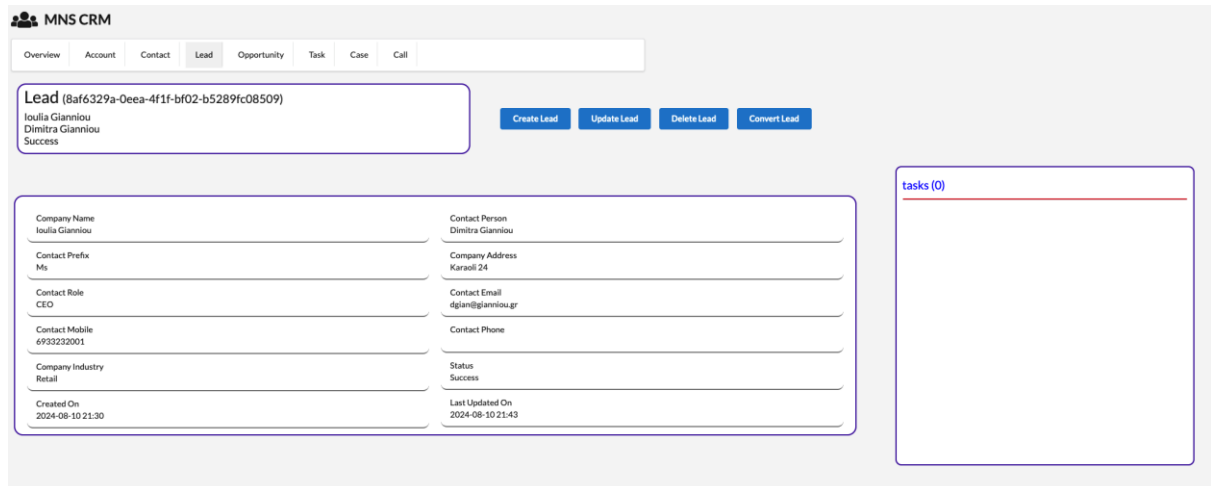


Figure 2.27. Lead Entry page with Convert action visible

### Convert Lead

Are you sure you want to convert Lead "Ioulia Gianniou"

An Account, Contact and Opportunity will be created based on Lead data.



Figure 2.28. Convert Lead Modal example

### Convert Lead

Congratulations!

Lead "Ioulia Gianniou" was successfully converted

Created Account: Ioulia Gianniou  
Created Contact: Dimitra Gianniou  
Created Opportunity: Ioulia Gianniou Opportunity



Figure 2.29. Convert Lead Modal Success example

In case an error is returned during those operations, an informative message-notification will appear on the top right of the screen. Such a case would be inserting a record with incomplete required data or updating a record to that state.

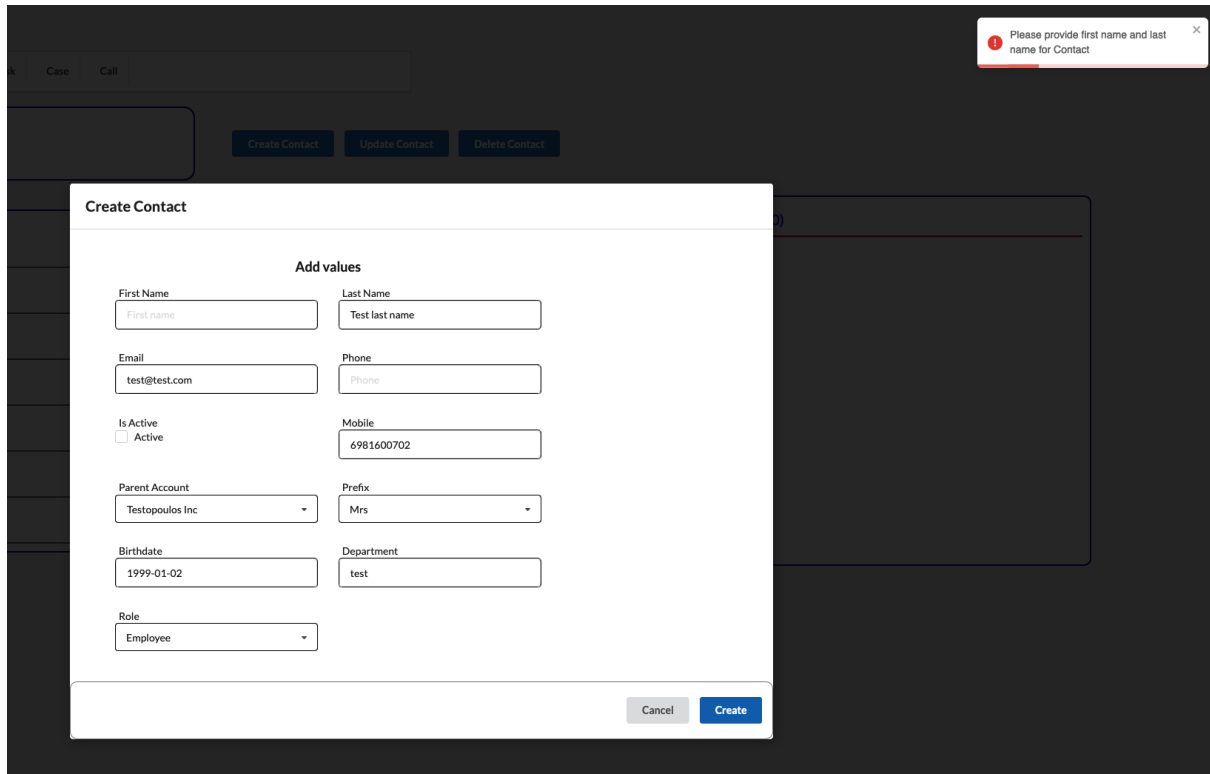


Figure 2.30. Data validation error for Contact example

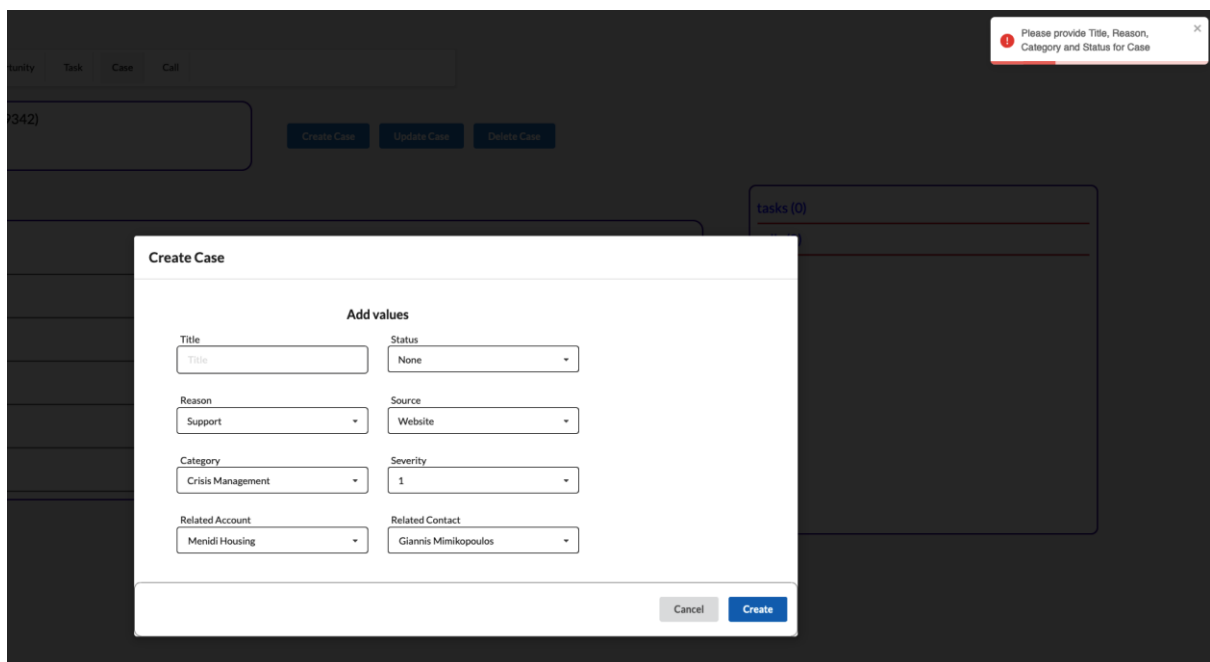


Figure 2.31. Data validation error for Case example

The create and update modals included fields that are different from TextInput. These are the Comboboxes, Checkboxes and EntryPointers. The Comboboxes's values are dynamically set by performing a request to the back-end table "Config\_cfg". The EntryPoint is special because it visualizes database entries as Combobox options by performing a request to recently updated records of the Entity it "points" at. The use of EntryPoint components is targeted for handling

parent relationships of the present entry of the Entry page. The EntryPoint component is also generic and reusable for all entities.

### Add values

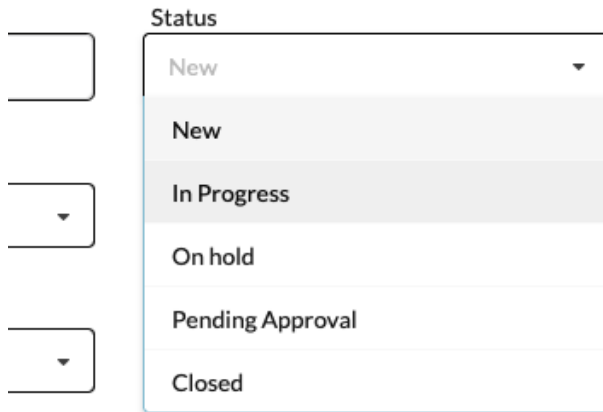


Figure 2.32. Dynamically defined Combobox options example

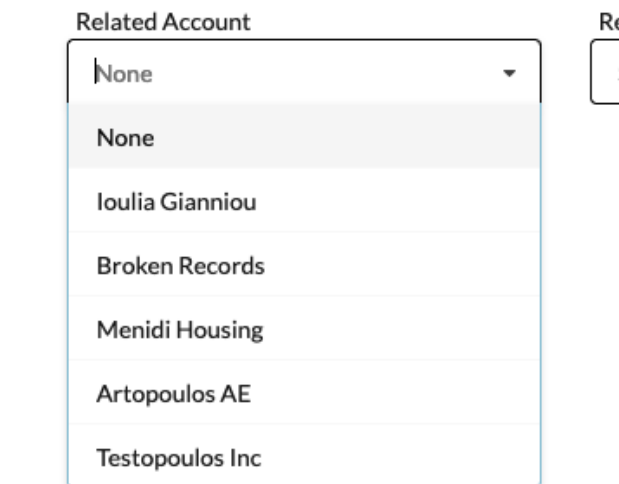


Figure 2.33. Entry pointer with Account record options example

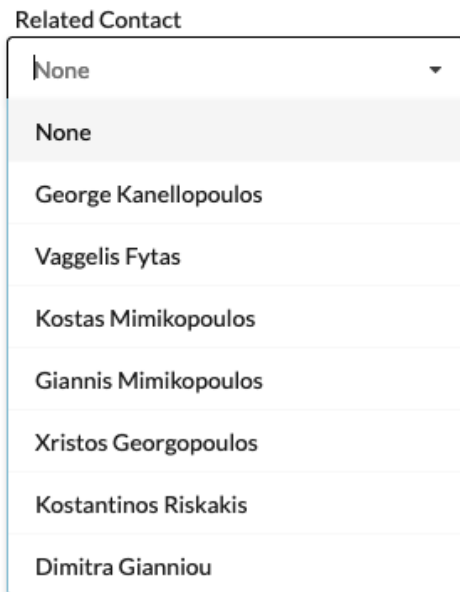


Figure 2.34. Entry pointer with Contact record options example

```

/* use functional form of state update to avoid race conditions,
   react state updates are batched and asynchronous */
function parseOptions(records){
  setOptionRecords(prevOptions => {
    const existingIds = new Set(prevOptions.map(option => option.key));
    const newOptions = records.filter(record => !existingIds.has(record.id))
      .map(record => (
        {
          key: record.id,
          value: record.id,
          text: getComboBoxOptionDisplayLabel(record)
        }
      ));
    return [
      ...prevOptions,
      ...newOptions
    ];
  });
}

```

Figure 2.35. Entry Pointer option parsing mechanism implementation

```

const fetchSampleRecords = async () => {
  const records = await RequestService.getLastModifiedRecords(entityName, 5, "modified", "desc");
  if(records){
    parseOptions(records);
    setLoading(false);
  }
};

```

Figure 2.36. Entry Pointer dynamic retrieval of database entries for display

Having seen the different components of the client application, we can view the complete component hierarchy of the React components.

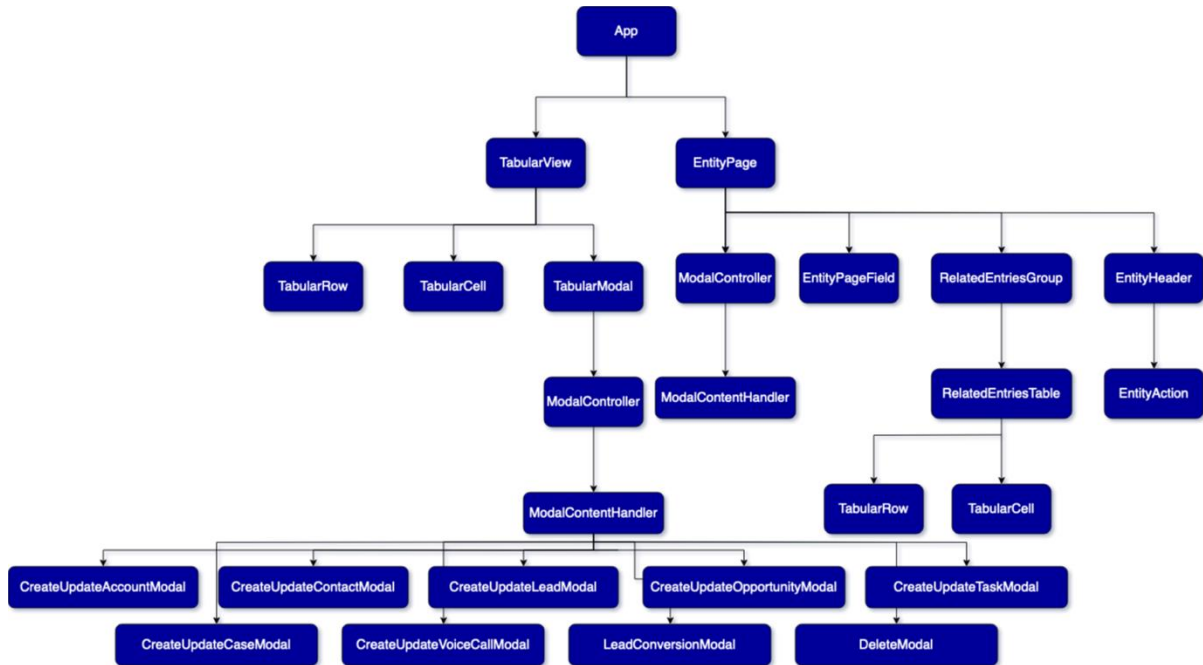


Figure 2.37. React Component Hierarchy

### 3.3.2 Design Pattern Implementation Presentation

The Java Springboot implementation is a web service, intended to provide an exposed REST API for the front-end application to use. Communication will be performed through JSON HTTP requests / responses. Through the API, the client will be able to perform CRUD operations on the CRM Entities and trigger business processes. The complete back-end application source code can be found at <https://github.com/emmprokak/mns-crm-backend-00>.

#### Implementation Architecture and General Design

The present architecture implements the concept of bidirectional references, where both parent and children entries will maintain references to the each-other. According to this design choice, the child contains an attribute that shares the same data-type with the parent entity and points to it, while the parent also maintains a list of entries that share the same data-type with the child entity. In this manner, a single request to the parent will also fetch its children entries and a single request to the child will include the parent entity with its static attributes.

All Entities of the CRM are annotated with the `@Entity` annotation and specify the tables and fields of the underlying database. Table and Columns names are set explicitly to avoid coupling database schema names with Java classes. All Entities that contain CRM data end in the postfix `“_ent”`, while the configuration tables will use the postfix `“_cfg”`.

The client interacts with the “Controller” layer of the back-end. The Controller layer is responsible for routing and redirecting the incoming requests to the “Service” layer, which constitutes the central component of the implementation and provides functions for all expected operations. Each Entity has its own Service class implementation, providing functions for inserting new entries as well as fetching, updating and deleting existing entries.

Even though the central Service layer is responsible for handling the Entity operations, it contains little to no business logic inside of it. All specific actions that are needed for the operations to take place are delegated into different components of the architecture.

More specifically, the handling of inserting and updating records is delegated to the InsertUpdateTrigger module. The trigger acts as a wrapper of actions that need to take place before successfully inserting / updating a database record. Calls are made to the ObjectMapper, who handles the static mapping of attribute values, the RelationshipMapper, who handles the bidirectional references between Entity entries and the ValidationProcessor, who ensures certain data requirements on the entries to be inserted / updated.

Another example would be the DeleteTrigger, that focuses on abstracting the entry deletion process while also removing all references from parent and children entries. This is a complicated and logic-heavy part of the implementation and needs to be handled in its own separate module.

All database operations are abstracted by the Data Access Layer interfaces that follow the “Repository” pattern. By extending the JpaRepository interface, functions for CRUD and query operations are handled by the framework and become available for each Entity.

When a request from the front-end is handled, a response is always sent back to the client. We avoided sending over the actual Entities, since this would tightly couple the front-end implementation with the Entity attributes of the database schema. For this reason, the Data Transfer Object pattern was utilized. Once the request of the client has been handled, the resulting entry (or list of entries) is transformed into an instance of a specific data class, which is then returned to the client.

Each Entity has its own DTOs that are always structured in an inheritance hierarchy. The “EntityDTOMinimal” DTOs contain only the core fields (non relationship fields) of each entry, while lacking the references to parent entries and the lists that contain the children entries. The “EntityDTOSimple” DTOs include the non relationship fields and the parent entry records, but lack the children related lists. Lastly, the “EntityDTOComplete” DTOs incorporate all non relationship fields, parent Entity entries and children related lists. This is to provide different options of information access to the front-end application. For instance, retrieving a list of entries for populating a Tabular View should not include lists of child records for each entry. On the contrary, rendering the Entry Page for a single entry can be based on one single request that also fetches parent and children entries.

Setting aside the Entities, there are also two Controllers for supporting the Configuration and Business Processes. The Configuration Controller is responsible for providing combobox options dynamically to the client. Standardized inputs like the account industry or case status can be modelled as comboboxes in the web app and these options are maintained in a separate configuration table of the back-end. The Configuration Controller is responsible for fetching the proper combobox options and returning them to the client for rendering. On the other hand, the Business Process Controller routes requests that trigger business processes to the appropriate logic to be executed.

### **Example Architecture – Opportunity Entity**

Requests are received on the OpportunityController and routed.

```

@RestController  ± emmprokak
@RequestMapping("/api/opportunity")
public class OpportunityController {

    @Autowired
    private OpportunityService opportunityService;

    @GetMapping("/{id}")  ± emmprokak
    public OpportunityDTO getOpportunity(@PathVariable String id){
        return opportunityService.getOpportunityById(id, getChildrenRelationships: false);
    }

    @GetMapping("/{id}/complete")  ± emmprokak
    public OpportunityDTO getOpportunityWithReIs(@PathVariable String id){
        return opportunityService.getOpportunityById(id, getChildrenRelationships: true);
    }

    @PostMapping("/new")  ± emmprokak
    public OpportunityDTO createOpportunity(@RequestBody Opportunity opportunity) throws DataValidationException {
        return opportunityService.insertOpportunity(opportunity);
    }

    @GetMapping("/all")  ± emmprokak
    public List<OpportunityDTO> getOpportunities(){
        return opportunityService.getAllOpportunities();
    }

    @PutMapping("/{id}")  ± emmprokak
    public OpportunityDTO updateOpportunity(@PathVariable String id, @RequestBody Opportunity opportunity) throws DataValidationException {
        return opportunityService.updateOpportunity(id, opportunity);
    }

    @DeleteMapping("/{id}")  ± emmprokak
    public boolean deleteOpportunity(@PathVariable String id) { return opportunityService.deleteOpportunityById(id); }
}

```

**Figure 3.1. Implementation of OpportunityController**

The OpportunityService class encapsulates the operations that need to be performed. The Data Access Layer (Repositories) is used for database operations like queries and the Insert Update Trigger / Delete Trigger handle the entry saving and deletion process respectively.

```

public OpportunityDTO getOpportunityById(String id, boolean getChildrenRelationships){ 2 usages  ± emmprokax
    Optional<Opportunity> opptyOptional = opportunityRepository.findById(id);

    if(!opptyOptional.isPresent()){
        throw new RuntimeException(
            HttpStatus.BAD_REQUEST,
            ErrorMessageUtility.getEntityNotFoundBySpecifier(
                Constants.Entity.OPPORTUNITY,
                Constants.Specifier.ID
            )
        );
    }

    if(getChildrenRelationships){
        return opptyOptional.get().toDTOComplete();
    }

    return opptyOptional.get().toDTOSimple();
}

public OpportunityDTO insertOpportunity(Opportunity opportunity) throws DataValidationException { 1 usage  ± emmprokax
    Opportunity opptyToInsert = new Opportunity();
    opptyToInsert = insertUpdateTrigger.handleOpportunityEntry(opportunity, opptyToInsert, isInsert: true);

    return opportunityRepository.save(opptyToInsert).toDTOSimple();
}

public List<OpportunityDTO> getAllOpportunities(){ 1 usage  ± emmprokax
    List<Opportunity> opptyList = opportunityRepository.findAll();

    if(opptyList.size() <= 0){
        return new ArrayList<>();
    }

    return ListConverter.convertEntitiesToDTOList(opptyList, Constants.DTO.CONVERT_TO_DTO_SIMPLE);
}

```

Figure 3.2. Opportunity Service class fragment

```

public Opportunity handleOpportunityEntry(Opportunity source, Opportunity target, boolean isInsert) throws DataValidationException {
    opportunityValidationProcessor.beforeSaveProcessing(source);
    target = ObjectMapper.mapOpportunityFields(source, target);
    target = relationshipMapper.mapOpportunityParents(source, target, isInsert);

    return target;
}

```

Figure 3.3. Handling an Opportunity entry inside the Insert Update Trigger

On the definition of an Entity there are the attributes-fields, constructors and functions that are implemented from the present interfaces, which will be discussed later in detail.



```

@Entity
@Table(name="Opportunity_ent")
public class Opportunity implements Sendable<OpportunityDTO>, DataEntity, ChildEntity, ParentEntity {
    @Id
    @GeneratedValue
    private String id;

    @ManyToOne
    @JoinColumn(name = "account_id")
    private Account relatedAccount;

    @Column(name="account_id_txt")
    private String relatedAccountId;

    @OneToMany(cascade = CascadeType.ALL)
    private List<Task> tasks;

    @Column(name="title")
    private String title;

    @Column(name="status")
    private String status;

    @Column(name="type")
    private String type;

    @Column(name="comments")
    private String comments;

    @Column(name="description")
    private String description;

    @Column(name="expected_revenue")
    private double expectedRevenue;
}

```

Figure 3.4. Declaration of the Opportunity Entity

## Quality Characteristics

For the present implementation, emphasis was put into ensuring software quality so that the implementation can be maintained and extended in a flexible manner. Its architecture is similar to other Spring applications, utilizing patterns such as Repositories for abstracting the data persistence layer, a Service Layer that encapsulates the core operations and Data Transfer Objects (DTOs).

## Reused Architecture patterns

The DAL is important for decoupling the database operations and queries from the rest of the implementation and promoting the separation of concerns. JPA Repositories offer an abstraction for interacting with the database, allowing for interchangeable data stores without the need for rewriting parts of the implementation logic. By making use of the DAL we can avoid hard coded queries into the business layer, which would tightly couple the application with the underlying data store implementation.

The Service Layer acts as an intermediary between the DAL and Controllers. It encapsulates the core business logic and operations, processing the incoming requests from the Controller layer

and interacting with the DAL both directly and indirectly. The separation of concerns is promoted because the Controller layer focus on handling the requests and routing, the Service Layer handles the business logic and the DAL abstracts the database operations. It is important to note that this “intermediary” role puts the Service Layer at the heart of the application architecture.

## DTOs

DTOs are data classes that are used for transmitting data back to the API consumer. They promote efficiency by collecting all requested data in a single data class, thus reducing the number of API calls to obtain the requested information, and promoting loose coupling between the data-model and the client application. In the scenario that an Entity was returned as is to the client, the web app’s implementation would be reliant on the database tables and columns not undergoing modifications in the future. Changes in the model of the back-end would entail refactoring for the web app as well, which can be very high cost if there are many consumers of the web service. Essentially, the DTOs abstract the underlying database schema of the API service for the client application.

In our implementation, we utilize a hierarchy of DTO classes for each Entity. The simplest are the Minimal DTOs, that contain only the static attributes of each entry, while missing the references to parent entries and the lists that contain the children entries. The Simple DTOs include the static attributes and the parent entry records, but lack the children related lists. Finally, Complete DTOs include all static attributes, parent entries and lists of children entries. This hierarchy provides the flexibility to use the appropriate DTO for each use-case. If we want to render the entry page for a single record, it makes sense to use the Complete DTO, since we also want to include links to parent entries and related children entries (and display their names). For retrieving a list of records, the Simple DTO is more appropriate, since it includes parent entries (which can be displayed in a table cell with the name of the parent) but does not include entire related lists of children per entry. Lastly, if a Simple DTO contains a reference to a parent entry, we don’t want to retrieve the parent’s parent entry, so we will use the Minimal DTO for the related entry of a Simple DTO.

“Static” in this context doesn’t not refer to the lack of object instantiation, but to the “fixed” data structure of the non-relationship fields. Static attributes include fields like Strings (Names, Comments, Categories, Industries), Dates (Birthdates, Case Closed Date) and Booleans (IsActive). Non static attributes refer to data-types that reference another CRM Entity (Account, Contact, Task etc.). In the case of non-static attributes, the value of the field can be an Entity object with varying state.

All Service Layer and, as a result, Controller Layer methods utilize a DTO supertype as a return type. This is to add flexibility and promote loose coupling. For example, new fourth type of DTO can potentially be added without the need for refactoring the existing functions of the Controller and Service layer. Methods can be easily updated to return multiple types of DTOs without the need for duplicated logic.

```

public AccountDTO getAccountById(String id, boolean getChildrenRelationships){ 2 usages  ⚡ emmprokax
    Optional<Account> accountOptional = accountRepository.findById(id);

    if(!accountOptional.isPresent()){
        throw new ResponseStatusException(
            HttpStatus.BAD_REQUEST,
            ErrorMessageUtility.getEntityNotFoundBySpecifier(
                Constants.Entity.ACCOUNT,
                Constants.Specifier.ID
            )
        );
    }

    if(getChildrenRelationships){
        return accountOptional.get().toDTOComplete();
    }

    return accountOptional.get().toDTOSimple();
}

```

Figure 3.5. Different DTO based on the case

## Entity Configuration

All CRM Entities implement the Sendable interface, which defines the methods to be implemented for getting converted into a DTO.

```

public interface Sendable<T> { 16 usages 7 implementations  ⚡ emmprokax
    T toDTOSimple(); 7 implementations  ⚡ emmprokax
    T toDTOComplete(); 7 implementations  ⚡ emmprokax
    T toDTOMinimal(); 7 implementations  ⚡ emmprokax
}

```

Figure 3.6. The Sendable interface

```

@Override  ⚡ emmprokax
public TaskDTO toDTOSimple() {
    return new TaskDTOSimple(dueDate, id, name, reason, status, type, relatedLead, relatedOpportunity, created, modified);
}

@Override  ⚡ emmprokax
public TaskDTO toDTOComplete() {
    return new TaskDTOComplete(dueDate, id, name, reason, status, type, relatedLead, relatedOpportunity, created, modified);
}

@Override  ⚡ emmprokax
public TaskDTO toDTOMinimal() {
    return new TaskDTOMinimal(dueDate, id, name, reason, status, type, relatedLead, relatedOpportunity, created, modified);
}

```

Figure 3.7. Sendable interface implementation on Task

With all Entities sharing the Sendable interface and providing implementation for its defined functions, we can create a generic function that converts a list of Entities to a list of EntityDTOs. This is necessary for actions fetching a list of entries to populate a Tabular View, where each entry needs to be converted into DTO to be sent over with an API response.

```

public class ListConverter {  emmprokakis
    public static <E extends Sendable<D>, D> List<D> convertEntitiesToDTOList(List<E> entities, int conversionType) {
        List<D> dtoList = new ArrayList<>();

        for (E entity : entities) {
            D dto;
            switch (conversionType) {
                case Constants.DTO.CONVERT_TO_DTO_MINIMAL:
                    dto = entity.toDTOMinimal();
                    break;
                case Constants.DTO.CONVERT_TO_DTO_SIMPLE:
                    dto = entity.toDTOSimple();
                    break;
                case Constants.DTO.CONVERT_TO_DTO_COMPLETE:
                    dto = entity.toDTOComplete();
                    break;
                default:
                    throw new IllegalArgumentException("Invalid conversion type");
            }
            dtoList.add(dto);
        }

        return dtoList;
    }
}

```

Figure 3.8. List Converter Generic implementation

```

public List<VoiceCallDTO> getAllVoiceCalls(){ 1 usage  emmprokakis *
    List<VoiceCall> voiceCallList = voiceCallRepository.findAll();

    if(voiceCallList.size() <= 0){
        return new ArrayList<>();
    }

    return ListConverter.convertEntitiesToDTOList(voiceCallList, Constants.DTO.CONVERT_TO_DTO_SIMPLE);
}

```

Figure 3.9. Voice Call utilizing generic list conversion to DTO

```

public AccountDTOComplete(String billingAddress, int clientRating, String companyName, String description, String id, String industry, boolean isActive, String parent, relatedLead, revenue, type,
    super(billingAddress, clientRating, companyName, description, id, industry, isActive, parent, relatedLead, revenue, type,
    this.children = ListConverter.convertEntitiesToDTOList(accounts, Constants.DTO.CONVERT_TO_DTO_MINIMAL);
    this.contacts = ListConverter.convertEntitiesToDTOList(contacts, Constants.DTO.CONVERT_TO_DTO_MINIMAL);
    this.cases = ListConverter.convertEntitiesToDTOList(cases, Constants.DTO.CONVERT_TO_DTO_MINIMAL);
    this.calls = ListConverter.convertEntitiesToDTOList(calls, Constants.DTO.CONVERT_TO_DTO_MINIMAL);
    this.opportunities = ListConverter.convertEntitiesToDTOList(opportunities, Constants.DTO.CONVERT_TO_DTO_MINIMAL);
}

```

Figure 3.10. AccountDTOComplete making use of generic list conversion for related children

The CRM Entity classes define the attributes and relationships between Entities. They construct the database schema, determining the tables, columns and relationships. An Entity's class structure includes the class declaration, the class field members, constructors, the interface implemented methods and getters-setters. The table name of each CRM Entity is made up by the entity name and the postfix “\_ent”. All column names are explicitly defined and follow the snake case format for their naming scheme. The names of tables and columns are explicitly specified in order to decouple the database implementation for the Java classes of the web service.

All Entities implement a couple of interfaces. The `Sendable<T>` interface defines all entities that can be “sent over” to a different application, as part of an API response. The generic parameter “T” refers to the DTO mapped to the object implementing the interface. For instance, in the case of `Account`, the implementation would be “class `Account` implements `Sendable<AccountDTO>`”. The `DataEntity` interface defines no methods to be implemented -as of now- and is used to keep a polymorphic reference to a list of different type of Entity entries. Lastly, the `ParentEntity` and `ChildEntity` interfaces define methods for handling the updates to an Entity’s parent and children relationships. An Entity can be both a `ParentEntity` and a `ChildEntity` and these interfaces are necessary to handle the bidirectional reference updates in an agnostic and generic way. This generic mechanism will be explored in more details below. Generic bidirectional updates allow for great flexibility, extendibility and reduced duplicated logic.

```
@Override  ± emmprokak
public <P> P getParent(Class<P> entityType) {
    if(entityType == Lead.class){
        return (P) relatedLead;
    }

    if(entityType == Opportunity.class){
        return (P) relatedOpportunity;
    }

    return null;
}

@Override  ± emmprokak
public <P> void setParent(Class<P> entityType, P parent) {
    if(entityType == Lead.class){
        this.relatedLead = (Lead) parent;
        if(parent != null){
            this.relatedLeadId = ((Lead) parent).getId();
        }else{
            this.relatedLeadId = null;
        }
    }

    if(entityType == Opportunity.class){
        this.relatedOpportunity = (Opportunity) parent;
        if(parent != null){
            this.relatedOpportunityId = ((Opportunity) parent).getId();
        }else{
            this.relatedOpportunityId = null;
        }
    }
}
```

Figure 3.11. `ChildEntity` implementation on `Task`

```
@Override 1 usage  ⚠ emmprokak
public <C> List<C> getChildrenEntities(Class<C> childType) {
    if (childType == Task.class) {
        return (List<C>) tasks;
    }

    return new ArrayList<>();
}

@Override 1 usage  ⚠ emmprokak
public <C> void addChild(Class<C> childType, C child) {
    if (childType == Task.class) {
        tasks.add((Task) child);
    }
}

@Override 1 usage  ⚠ emmprokak
public <C> void removeChild(Class<C> childType, C child) {
    if (childType == Task.class) {
        tasks.remove((Task) child);
    }
}
```

Figure 3.12. ParentEntity implementation on Opportunity

```

@Transactional 9 usages  ⚡ emmprokax *
public <P extends ParentEntity, C extends ChildEntity> C handleParentChildRelationship(
    C reqChild,
    C childToBeUpdated,
    JpaRepository<P, String> parentRepository,
    JpaRepository<C, String> childRepository,
    Class<P> parentType,
    Class<C> childType,
    Boolean isInsert
) {
    String newParentId = reqChild.getParentId(parentType);
    String oldParentId = childToBeUpdated.getParentId(parentType);

    if (oldParentId != null && !StringUtil.stringsAreEqual(oldParentId, newParentId)) {
        Optional<P> oldParentOptional = parentRepository.findById(oldParentId);
        if (oldParentOptional.isPresent()) {
            P oldParent = oldParentOptional.get();
            oldParent.removeChild(childType, childToBeUpdated);
            parentRepository.save(oldParent);
        }
    }

    if (newParentId == null) {
        childToBeUpdated.setParent(parentType, parent: null);
        return childRepository.save(childToBeUpdated);
    }

    Optional<P> newParentOptional = parentRepository.findById(newParentId);
    if (!newParentOptional.isPresent()) {
        throw new ResponseStatusException(
            HttpStatus.BAD_REQUEST,
            "Parent entity not found"
        );
    }

    P newParent = newParentOptional.get();
    childToBeUpdated.setParent(parentType, newParent);

    if (!newParent.getChildrenEntities(parentType).contains(childToBeUpdated)) {
        newParent.addChild(childType, childToBeUpdated);
    }

    if (isInsert) {
        childRepository.save(childToBeUpdated);
    }

    parentRepository.save(newParent);
    return childRepository.save(childToBeUpdated);
}

```

Figure 3.13. Generic Handling of bidirectional reference updates

### Insert Update Trigger

The Service layer offers functions that support all the core operations. It breaks those actions into smaller tasks and delegates these tasks to the appropriate module. Such is the case with the Insert Update Trigger (IUT), who is responsible handling the data validation, static attribute mapping and relationship mapping for an Entity entry before saving it to the database. The IUT contains one function for each CRM Entity in order to handle its insert/update action.

```

@Autowired
private ValidationHandler validationHandler;

public Account handleAccountEntry(Account source, Account target) throws DataValidationException {
    validationHandler.validate(source);
    target = ObjectMapper.mapAccountFields(source, target);
    target = relationshipMapper.mapAccountParents(source, target);

    return target;
}

public Contact handleContactEntry(Contact source, Contact target, boolean isInsert) throws DataValidationException {
    validationHandler.validate(source);
    target = ObjectMapper.mapContactFields(source, target);
    target = relationshipMapper.mapContactParents(source, target, isInsert);

    return target;
}

public Lead handleLeadEntry(Lead source, Lead target) throws DataValidationException {
    validationHandler.validate(source);
    target = ObjectMapper.mapLeadFields(source, target);
    // lead has no parents
    return target;
}

```

Figure 3.14. Insert Update Trigger implementation for Account, Contact and Lead

The first action of the IUT is to validate the data for the respective Entity entry. This method is generic and agnostic to the Entity type of the entry provided. For this part of the implementation (data validation) the Template Method design pattern was used. By making use of the Template Method pattern, we can ensure the core structure of the algorithm remains the same, while we defer the algorithm step implementation to subclasses. We can ensure that the data validation method always runs before the beforeSave method and the inheriting classes can only provide an implementation for those methods, but not change the order method calls or avoid their execution.

```

public abstract class ValidationTemplate<E> {
    public final void beforeSaveProcessing(E entry) throws DataValidationException {
        validate(entry);
        beforeSave(entry);
    }

    protected abstract void validate(E entry) throws DataValidationException;

    protected void beforeSave(E entry){
        return;
    }
}

```

Figure 3.15. Validation Class supertype



The ValidationTemplate class was made to be extended by the individual ValidationProcessor classes. We decided to move forward with one ValidationProcessor class per Entity, so that the logic will be separate for different entity types and, thus, easy to maintain and extend. Each of these classes implements the ValidationTemplate interface and provides an implementation for the “validate” function. Said function is declared protected so that overriding is possible. This is contrast with the beforeSaveProcessing method, which is declared final for its implementation to be unmodifiable. For the calling environment to interact with the ValidationHandler class should be made, which offers a generic method for Entities that dynamically maps the proper ValidationProcessor based on the class type key.

```

@Component 2 usages new *
public class ValidationHandler {

    private final Map<Class<?>, ValidationTemplate<?>> validators = new HashMap<>(); 8 usages

    public ValidationHandler() { new *
        validators.put(Account.class, new AccountValidationProcessor());
        validators.put(Contact.class, new ContactValidationProcessor());
        validators.put(Lead.class, new LeadValidationProcessor());
        validators.put(Task.class, new TaskValidationProcessor());
        validators.put(Opportunity.class, new OpportunityValidationProcessor());
        validators.put(VoiceCall.class, new VoiceCallValidationProcessor());
        validators.put(Case.class, new CaseValidationProcessor());
    }

    public <T> void validate(T entity) throws DataValidationException { new *
        ValidationTemplate<T> validator = (ValidationTemplate<T>) validators.get(entity.getClass());

        if (validator != null) {
            validator.beforeSaveProcessing(entity);
        }else{
            throw new IllegalArgumentException("No validator found for " + entity.getClass().getSimpleName());
        }
    }
}

```

Figure 3.16. Generic ValidationHandler for calling Entity-specific ValidationProcessor classes

```

@Component 1 usage emmprokax
public class AccountValidationProcessor extends ValidationTemplate<Account> {
    @Override 1 usage emmprokax
    protected void validate(Account entry) throws DataValidationException {
        if(StringUtil.stringIsEmptyOrNull(entry.getCompanyName()) || StringUtil.stringIsEmptyOrNull(entry.getVat())){
            throw new DataValidationException("Please provide Company Name and VAT");
        }
    }
}

```

Figure 3.17. Implementation of the Account Validation

```
@Component 1 usage 1 emmprokak
public class LeadValidationProcessor extends ValidationTemplate<Lead> {
    @Override 1 usage 1 emmprokak
    protected void validate(Lead entry) throws DataValidationException {
        if(StringUtil.stringIsEmptyOrNull(entry.getCompanyName())){
            throw new DataValidationException("Please provide Company Name for Lead");
        }
    }
}
```

**Figure 3.18. Implementation of the Lead Validation**

With this pattern, we achieve a positive impact for the implementation. For instance, modifications on the criteria of data validations can be made directly to the Entity-specific classes without the need for modifications to other Entities or the rest of the codebase. Actions before or after the validation method, can be easily performed by updating the parent `ValidationTemplate` class with new methods and providing default implementations for those methods. In addition, adding validation logic for a new Entity can be done by creating a new `ValidationProcessor` class and implementing the `ValidationTemplate` interface, without the need to modify existing validation logic in existing classes. Setting aside the Entity-specific operations, an action can be forced on all Entities by just updating the parent `ValidationTemplate` class and the `beforeSaveProcessing` method. Overall, the application of this pattern boosts maintainability, flexibility and extensibility in the IUT module.

Having processed the request entry for enforcing data criteria, the IUT moves on to mapping the static attributes of the entry. The Object Mapper (OM) is a class that handles the mapping of the static attributes between the entry provided in the request (source) to the entry already present in the database (target). For inserting records, the target is an empty entry of that Entity which was just instantiated. The OM offers a function for every Entity and makes use of the Builder design pattern to properly construct the resulting Entity entry. The Builder pattern abstracts the instantiation process from the calling environment, targeting the construction of complex objects with varying state, while promoting the separation of concerns. With the use of Builder, we can abstract the instance creation process from the calling environment code (Object Mapper in this case) and avoid hard-coding certain logic. An example of hard-coding logic would be a null check for an assignment that would otherwise throw an Exception as this was the case for Date fields.

```

public static Lead mapLeadFields(Lead source, Lead target){ 1 usage  ± emmprokax
    return new LeadBuilder(target)
        .setCompanyAddress(source.getCompanyAddress())
        .setCompanyIndustry(source.getCompanyIndustry())
        .setCompanyName(source.getCompanyName())
        .setContactEmail(source.getContactEmail())
        .setContactMobile(source.getContactMobile())
        .setContactPhone(source.getContactPhone())
        .setContactRole(source.getContactRole())
        .setContactPrefix(source.getContactPrefix())
        .setStatus(source.getStatus())
        .setContactPerson(source.getContactPerson())
        .build();
}

public static Opportunity mapOpportunityFields(Opportunity source, Opportunity target){ 1 usage  ± emmprokax
    return new OpportunityBuilder(target)
        .setStatus(source.getStatus())
        .setTitle(source.getTitle())
        .setExpectedRevenue(source.getExpectedRevenue())
        .setComments(source.getComments())
        .setType(source.getType())
        .setDescription(source.getDescription())
        .build();
}

public static Task mapTaskFields(Task source, Task target){ 1 usage  ± emmprokax
    return new TaskBuilder(target)
        .setStatus(source.getStatus())
        .setDueDate(source.getDueDate())
        .setName(source.getName())
        .setReason(source.getReason())
        .setStatus(source.getStatus())
        .setType(source.getType())
        .build();
}

```

Figure 3.19. Object Mapper using Builder Pattern

```

public ContactBuilder setBirthdate(Date birthdate) { 1 usage  ± emmprokax
    if(birthdate != null){
        contact.setBirthdate(birthdate);
    }
    return this;
}

```

Figure 3.20. Contact Builder performing null check before assignment

All Builder classes share a common interface that defines the build method and each Entity has its own Builder class.

```

public interface EntityBuilder<T extends DataEntity> {
    T build(); 7 implementations  ± emmprokax
}

```

Figure 3.21. The Entity Builder interface

```

public class TaskBuilder implements EntityBuilder<Task>{ 6 usages  ± emmprokax
    private Task task; 8 usages

    @Override  ± emmprokax
    public Task build() {
        return task;
    }

    public TaskBuilder(){ no usages  ± emmprokax
        this.task = new Task();
    }

    public TaskBuilder(Task task){ 1 usage  ± emmprokax
        this.task = task;
    }

    public TaskBuilder setName(String name) { ± emmprokax
        task.setName(name);
        return this;
    }

    public TaskBuilder setReason(String reason) { ± emmprokax
        task.setReason(reason);
        return this;
    }
}

```

Figure 3.22. Task Builder class fragment

Having concluded with the static attribute mappings, the IUT proceeds with the relationship (dynamic) mappings which are handled by a module named Relationship Mapper (RM).

```

public Case handleCaseEntry(Case source, Case target, boolean isInsert) throws DataValidationException { 2 usages  ± emmprokax
    caseValidationProcessor.beforeSaveProcessing(source);
    target = ObjectMapper.mapCaseFields(source, target, isInsert);
    target = relationshipMapper.mapCaseParents(source, target, isInsert);

    return target;
}

public VoiceCall handleVoiceCallEntry(VoiceCall source, VoiceCall target, boolean isInsert) throws DataValidationException { 2 usages  ± emmprokax
    voiceCallValidationProcessor.beforeSaveProcessing(source);
    target = ObjectMapper.mapVoiceCallFields(source, target);
    target = relationshipMapper.mapVoiceCallParents(source, target, isInsert);

    return target;
}

```

Figure 3.23. Insert Update Trigger for Cases and VoiceCalls

The RM is responsible for handling the bidirectional references on an Entity entry and its parents. RM provides a function for each Entity and inside the function the generic handleParentChildRelationship method is called for every parent of the entry. A comparison is made to see if the parentId field has been modified in comparison to what is currently present in the database. If the parentId has changed (to null or another value) the generic handleParentChildRelationship method is called for that parent and request entry pair. This check is valuable before entering the method, since this is a query-heavy part of the implementation.

```

public Opportunity mapOpportunityParents(Opportunity reqOppty, Opportunity opptyToBeUpdated, Boolean isInsert){ 1 usage  ± emmprokak
    if(!StringUtil.stringsAreEqual(reqOppty.getRelatedAccountId(), opptyToBeUpdated.getRelatedAccountId())){
        opptyToBeUpdated = relHandlerHelper.handleParentChildRelationship(
            reqOppty,
            opptyToBeUpdated,
            accountRepository,
            opportunityRepository,
            Account.class,
            Opportunity.class,
            isInsert
        );
    }

    return opptyToBeUpdated;
}

public Task mapTaskParents(Task reqTask, Task taskToBeUpdated, Boolean isInsert){ 1 usage  ± emmprokak
    if(!StringUtil.stringsAreEqual(reqTask.getRelatedLeadId(), taskToBeUpdated.getRelatedLeadId())){
        taskToBeUpdated = relHandlerHelper.handleParentChildRelationship(
            reqTask,
            taskToBeUpdated,
            leadRepository,
            taskRepository,
            Lead.class,
            Task.class,
            isInsert
        );
    }

    if(!StringUtil.stringsAreEqual(reqTask.getRelatedOpportunityId(), taskToBeUpdated.getRelatedOpportunityId())){
        taskToBeUpdated = relHandlerHelper.handleParentChildRelationship(
            reqTask,
            taskToBeUpdated,
            opportunityRepository,
            taskRepository,
            Opportunity.class,
            Task.class,
            isInsert
        );
    }

    return taskToBeUpdated;
}

```

**Figure 3.24.** The generic `handleParentChildRelationship` method is called for `Opportunity` and `Task`

This module (RM) does not use any of the 26 Gang of Four patterns. However, it utilizes a lot of ideas that go into the design patterns for what makes a good implementation. The handling of the parent-child bidirectional references is made generic in order to avoid duplicate logic. We are going to see how that evolved in more detail on the version of the web service that doesn't implement the design pattern ideas and practices. (Section 3.3.3)

The method utilizes Generics, Bounded Types, class type keys and interfaces to be able to manage complex bidirectional references between Entities with varying relationships. The method parameter types, return types, arguments and body statements make heavy use of generics. These are dynamically set data types of the variables, which are not known at compile-time. The generics P(arent) and C(hild) are upper bounded by the interfaces `ParentEntity` and `ChildEntity` respectively, which allows us to use their defined functions, since all subtypes of them will implement them. The class type keys are used as function arguments so that they can be passed down to the Entity methods that will handle them accordingly.

The logic of the function is the following: if the parent entry has changed and the old parent entry existed, remove the references of the present entry from its children. If the new parent id is empty, remove all references from the previous parent on the child and return out of the function.

Alternatively, if a new parent is present, update both the new parent and the present entry to point to each other.

```

@Transactional 9 usages  ⤴ emmprokax *
public <P extends ParentEntity, C extends ChildEntity> C handleParentChildRelationship(
    C reqChild,
    C childToBeUpdated,
    JpaRepository<P, String> parentRepository,
    JpaRepository<C, String> childRepository,
    Class<P> parentType,
    Class<C> childType,
    Boolean isInsert
) {
    String newParentId = reqChild.getParentId(parentType);
    String oldParentId = childToBeUpdated.getParentId(parentType);

    if (oldParentId != null && !StringUtil.stringsAreEqual(oldParentId, newParentId)) {
        Optional<P> oldParentOptional = parentRepository.findById(oldParentId);
        if (oldParentOptional.isPresent()) {
            P oldParent = oldParentOptional.get();
            oldParent.removeChild(childType, childToBeUpdated);
            parentRepository.save(oldParent);
        }
    }

    if (newParentId == null) {
        childToBeUpdated.setParent(parentType, parent: null);
        return childRepository.save(childToBeUpdated);
    }

    Optional<P> newParentOptional = parentRepository.findById(newParentId);
    if (!newParentOptional.isPresent()) {
        throw new ResponseStatusException(
            HttpStatus.BAD_REQUEST,
            "Parent entity not found"
        );
    }

    P newParent = newParentOptional.get();
    childToBeUpdated.setParent(parentType, newParent);

    if (!newParent.getChildrenEntities(parentType).contains(childToBeUpdated)) {
        newParent.addChild(childType, childToBeUpdated);
    }

    if (isInsert) {
        childRepository.save(childToBeUpdated);
    }

    parentRepository.save(newParent);
    return childRepository.save(childToBeUpdated);
}

```

Figure 3.25. Generic parent-child reference handling

```

@Override 1 usage  emmprokak
public <C> List<C> getChildrenEntities(Class<C> childType) {
    // switch statement not supported for typeKey as of Java 21
    if (childType == Account.class) {
        return (List<C>) children;
    }

    if (childType == Contact.class) {
        return (List<C>) contacts;
    }

    if (childType == Case.class) {
        return (List<C>) cases;
    }

    if (childType == VoiceCall.class) {
        return (List<C>) calls;
    }

    if (childType == Opportunity.class) {
        return (List<C>) opportunities;
    }

    return new ArrayList<>();
}

```

Figure 3.26. ParentEntity method implemented for Account Entity

```

@Override  emmprokak
public <P> void setParent(Class<P> entityType, P parent) {
    if(entityType == Lead.class){
        this.relatedLead = (Lead) parent;
        if(parent != null){
            this.relatedLeadId = ((Lead) parent).getId();
        }else{
            this.relatedLeadId = null;
        }
    }

    if(entityType == Opportunity.class){
        this.relatedOpportunity = (Opportunity) parent;
        if(parent != null){
            this.relatedOpportunityId = ((Opportunity) parent).getId();
        }else{
            this.relatedOpportunityId = null;
        }
    }
}

```

Figure 3.27. ChildEntity method implemented for Task

In this manner, we are able to dynamically and generically handle the parent child bidirectional references. This is a complex and logic-heavy part of the implementation that can be used by all existing and future Entities of the CRM web service. This is great for flexibility, extendibility and code reusability within the MSN CRM back-end.

## Delete Trigger

The Service layer houses operations for entry deletions and delegates entry deletion logic to the Delete Trigger (DT) module. For this first version of MNS CRM it was decided to not cascade the delete operation to child entries. In essence, this means that deleting an Account entry will not get rid of its children entities, but they will become orphan records from that point on. The job of the Delete Trigger is to successfully remove all reference to the pending-to-be-deleted record, so that the delete call of the JPARepository will not trigger a database exception. By removing all references of an entry from other related database records, we can safely proceed with its deletion.

```
public boolean deleteAccountById(String id){ 1 usage  ⚡ emmprkak
    Optional<Account> accountOptional = accountRepository.findById(id);

    if(!accountOptional.isPresent()){
        throw new ResponseStatusException(
            HttpStatus.BAD_REQUEST,
            ErrorMessageUtility.getEntityNotFoundBySpecifier(
                Constants.Entity.ACCOUNT,
                Constants.Specifier.ID
            )
        );
    }

    Account accToDelete = accountOptional.get();
    deleteTrigger.handleReferenceDeletion(accToDelete);
    accountRepository.delete(accToDelete);

    return true;
}
```

Figure 3.28. Service Layer delete method of Account



```

public boolean deleteTaskById(String id){ 1 usage  ⤴ emmprokak
    Optional<Task> taskOptional = taskRepository.findById(id);

    if(!taskOptional.isPresent()){
        throw new ResponseStatusException(
            HttpStatus.BAD_REQUEST,
            ErrorMessageUtility.getEntityNotFoundBySpecifier(
                Constants.Entity.TASK,
                Constants.Specifier.ID
            )
        );
    }

    deleteTrigger.handleReferenceDeletion(taskOptional.get());
    taskRepository.delete(taskOptional.get());

    return true;
}

```

Figure 3.29. Service Layer delete method of Task

The Delete Trigger offers the `handleReferenceDeletion` method, which is generic and reusable for all Entities. It manages to remove all active references of the entry-to-be-deleted from other Entity entries. This functionality is made possible with the application of the Strategy design pattern. The latter is great for encapsulating different behaviours in classes defined by a common interface and dynamically utilizing each of them at runtime according to the context. The actual implementation of the “strategy” is not known at compile-time and the Strategy pattern accomplishes loose coupling between the implementation logic and the invoking class, making it more modular and flexible. Each Entity has its own `DeletionHandler` class that implements the `DeletionHandler` interface.

```

public interface DeletionHandler<T> { 20 usages 7 implementations  ⤴ emmprokak
    public T delete(T entity); 7 implementations  ⤴ emmprokak
    public T handleParentReferences(T entity); 7 usages 7 implementations  ⤴ emmprokak
    public T handleChildReferences(T entity); 7 usages 7 implementations  ⤴ emmprokak
}

```

Figure 3.30. Deletion Handler Interface

Having separate concrete `DeletionHandler` classes for each Entity allows us to differentiate how the cascading deletion works, if we decide to add it to the future. The `delete` method groups the `handleParentReferences` and the `handleChildReferences` methods together. Method `handleParentReferences` focuses on removing the present entry references from the related lists of parent entries, while `handleChildReferences` will query all children where their parent id is equal to the present entry id and clear that field.

```
@Override / usages emmprokak
public Case handleParentReferences(Case caseEntry){
    if(!StringUtil.isEmptyOrNull(caseEntry.getRelatedAccountId())){
        Optional<Account> accountOptional = accountRepository.findById(caseEntry.getRelatedAccountId());

        if(!accountOptional.isPresent()){
            throw new RuntimeException(
                HttpStatus.BAD_REQUEST,
                ErrorMessageUtility.getEntityNotFoundBySpecifier(
                    Constants.Entity.ACCOUNT,
                    Constants.Specifier.ID
                )
            );
        }

        accountOptional.get().getCases().remove(caseEntry);
        accountRepository.save(accountOptional.get());
    }

    if(!StringUtil.isEmptyOrNull(caseEntry.getRelatedContactId())){
        Optional<Contact> contactOptional = contactRepository.findById(caseEntry.getRelatedContactId());

        if(!contactOptional.isPresent()){
            throw new RuntimeException(
                HttpStatus.BAD_REQUEST,
                ErrorMessageUtility.getEntityNotFoundBySpecifier(
                    Constants.Entity.CONTACT,
                    Constants.Specifier.ID
                )
            );
        }

        contactOptional.get().getCases().remove(caseEntry);
        contactRepository.save(contactOptional.get());
    }

    return caseEntry;
}
```

Figure 3.31. Implementation of handleParentReferences of Case

```
@Override 7 usages  ▲ emmprokak
public Account handleChildReferences(Account account){
    List<Account> accountList = accountRepository.findByParentId(account.getId());
    for(Account acc : accountList){
        acc.setParent(null);
        acc.setParentId(null);
        accountRepository.save(acc);
    }

    List<Contact> contactList = contactRepository.findByAccountId(account.getId());
    for(Contact con : contactList){
        con.setAccount(null);
        con.setAccountId(null);
        contactRepository.save(con);
    }

    List<Opportunity> opportunityList = opportunityRepository.findByRelatedAccountId(account.getId());
    for(Opportunity opp : opportunityList){
        opp.setRelatedAccount(null);
        opp.setRelatedAccountId(null);
        opportunityRepository.save(opp);
    }

    List<Case> caseList = caseRepository.findByRelatedAccountId(account.getId());
    for(Case c : caseList){
        c.setRelatedAccount(null);
        c.setRelatedAccountId(null);
        caseRepository.save(c);
    }

    return account;
}
```

**Figure 3.32. Implementation of handleChildReferences of Account**

All DeletionHandler classes are mapped by class type keys inside the DeletionHandlerConfig class. The latter is annotated as `@Configuration`, so that Springboot will inject the appropriate DeletionHandler class into the Context.

```
@Configuration
public class DeletionHandlerConfig {

    @Bean
    public Map<Class<?>, DeletionHandler<?>> deletionHandlers(AccountDeletionHandler accountHandler,
                                                             ContactDeletionHandler contactHandler,
                                                             OpportunityDeletionHandler opportunityHandler,
                                                             LeadDeletionHandler leadHandler,
                                                             TaskDeletionHandler taskHandler,
                                                             CaseDeletionHandler caseHandler,
                                                             VoiceCallDeletionHandler voiceCallHandler
    ) {
        Map<Class<?>, DeletionHandler<?>> handlers = new HashMap<>();
        handlers.put(Account.class, accountHandler);
        handlers.put(Contact.class, contactHandler);
        handlers.put(Opportunity.class, opportunityHandler);
        handlers.put(Lead.class, leadHandler);
        handlers.put(Task.class, taskHandler);
        handlers.put(Case.class, caseHandler);
        handlers.put(VoiceCall.class, voiceCallHandler);
        return handlers;
    }
}
```

**Figure 3.33. Configuration provided by the DeletionHandlerConfig class**

Thanks to this configuration and the flexibility provided by applying the Strategy design pattern, the implementation of the DeleteTrigger class is simple. The handleReferenceDeletion method accepts an Entity entry, gets the appropriate DeletionHandler class dynamically at runtime and, after a null check, calls the interface method delete, whose implementation is decided at that moment.

```

@Component 14 usages  ⚡ emmprokak *
public class DeleteTrigger {

    private final Map<Class<?>, DeletionHandler<?>> handlers; 2 usages

    @Autowired  ⚡ emmprokak
    public DeleteTrigger(Map<Class<?>, DeletionHandler<?>> handlers) {
        this.handlers = handlers;
    }

    @SuppressWarnings("unchecked") 7 usages  ⚡ emmprokak *
    public <T> void handleReferenceDeletion(T entity) {
        DeletionHandler<T> handler = (DeletionHandler<T>) handlers.get(entity.getClass());
        if (handler != null) {
            handler.delete(entity);
        } else {
            throw new IllegalArgumentException(
                "No handler found for Entity: " + entity.getClass().getName()
            );
        }
    }
}

```

Figure 3.34. Implementation of DeleteTrigger

## Business Processes

Having covered the way our Java application handles the Entity database operations, it's time to move on to the Business Processes. A Business Process Controller class is exposed to the client application for handling the routing of requests to the Business Process Service layer, which houses the core process operations.

```

@RestController  ⚡ emmprokak
@RequestMapping(Ⓜ"/api/process")
public class BusinessProcessController {

    @Autowired
    private BusinessProcessService businessProcessService;

    @PostMapping (Ⓜ"/lead-conversion/{leadId}")  ⚡ emmprokak
    public List<EntityDTO> convertLead(@PathVariable String leadId){
        return businessProcessService.convertLead(leadId);
    }

    @GetMapping (Ⓜ"/discount")  ⚡ emmprokak
    public ProcessOutputDTO clientDiscount(@RequestParam String accountId, @RequestParam double amount){
        return businessProcessService.getClientDiscount(accountId, amount);
    }
}

```

Figure 3.35. Business Process Controller implementation

```
@Service 2 usages  ⤴ emmprakak
public class BusinessProcessService {

    @Autowired
    private BusinessProcess businessProcess;

    public List<EntityDTO> convertLead(String leadId){ 1 usage  ⤴ emmprakak
        return businessProcess.leadConversion(leadId);
    }

    public ProcessOutputDTO getClientDiscount(String accountId, double amount){ 1 usage
        return businessProcess.clientDiscount(accountId, amount);
    }

}
```

**Figure 3.36. Business Process Service implementation**

The business logic specifics regarding the exposed processes are handled by the Business Process (BP) module, which encapsulates the business logic implementation. Our web service currently supports two business process, with more to come in the future: Lead Conversion and Discount Calculation.

The Lead Conversion is the process of converting a potential customer into an Account, a Contact and an Opportunity entry. By leading leads through the Sales Funnel and successfully reaching the end of the Lead status lifecycle, the potential customer can be modeled as its own Account Entity in the CRM, with staff becoming Contacts and the sales attempt becoming an Opportunity. Converting a Lead signifies the transition of an external party from someone that has heard about the company and might be interested in buying a product to someone who is actively interested and is waiting to receive an offer.

The implementation of the Lead Conversion process is handled by the leadConversion method. By breaking down the problem to smaller actions, in order to convert a Lead, we need to check if the lead exists, create the three individual entries with their static attributes mapped, related the created entries with the parent Lead and map the relationships between them. After creating all the new entries, a list of their DTOs will be returned to the Service Layer, the Controller Layer and, eventually, the client.

```

@Transactional 1 usage  ⤴ emmprokak *
public List<EntityDTO> leadConversion(String leadId) {

    Optional<Lead> leadOptional = LeadRepository.findById(leadId);

    if(!leadOptional.isPresent()){
        throw new RuntimeException(
            HttpStatus.BAD_REQUEST,
            ErrorMessageUtility.getEntityNotFoundBySpecifier(
                Constants.Entity.LEAD,
                Constants.Specifier.ID
            )
        );
    }

    Lead inputLead = leadOptional.get();
    Account acc = new Account();
    Contact con = new Contact();
    Opportunity opp = new Opportunity();

    LeadConversionInvoker invoker = new LeadConversionInvoker();

    invoker.addCommand(new CreateAccountCommand(inputLead, acc));
    invoker.addCommand(new CreateContactCommand(inputLead, con));
    invoker.addCommand(new CreateOpportunityCommand(inputLead, opp));
    invoker.addCommand(new MapLeadToChildrenCommand(inputLead, acc, con, opp, relationshipMapper, accountRepository));
    invoker.addCommand(new MapLeadConversionChildrenRelationships(acc,
        con, opp,
        relationshipMapper,
        contactRepository,
        opportunityRepository)
    );

    invoker.executeCommands();

    return Arrays.asList(acc.toDTOSimple(), con.toDTOSimple(), opp.toDTOSimple());
}

```

Figure 3.37. Lead Conversion method implementation

The implementation of lead conversion steps is materialized with the use of the Command design pattern. Through the Command pattern, we achieve decoupling the calling environment (in this case the BP module) from the actual implementation of each step. All Command objects share a common interface that defines an execute method and encapsulate a request. Existing command-step implementation can be easily performed without the need for altering the BP module and new commands can be added by creating new classes and implementing the Command interface. This allows for great modularity, maintainability and extensibility.

```

public interface Command { 8 usages 5 implementations  ⤴ emmprokak
    void execute(); 1 usage 5 implementations  ⤴ emmprokak
}

```

Figure 3.38. The Command interface

```
public class CreateAccountCommand implements Command{ 1 usage  ⤴ emmprokak
    private Lead lead; 2 usages
    private Account account; 3 usages

    public CreateAccountCommand(Lead lead, Account account){ 1 usage  ⤴ emmprokak
        this.lead = lead;
        this.account = account;
    }

    @Override 1 usage  ⤴ emmprokak
    public void execute() {
        account = ObjectMapper.mapLeadToAccount(lead, account);
    }
}
```

Figure 3.39. Create Account Command implementation

```
public class CreateOpportunityCommand implements Command{ 1 usage  ⤴ emmprokak
    private Lead lead; 2 usages
    private Opportunity opportunity; 3 usages

    public CreateOpportunityCommand(Lead lead, Opportunity opportunity){ 1 usage  ⤴ emmprokak
        this.lead = lead;
        this.opportunity = opportunity;
    }

    @Override 1 usage  ⤴ emmprokak
    public void execute() {
        opportunity = ObjectMapper.mapLeadToOpportunity(lead, opportunity);
    }
}
```

Figure 3.40. Create Opportunity Command implementation



```
public class MapLeadConversionChildrenRelationships implements Command{ 1 usage  ⚡ emmprokak *
    private Account account; 2 usages
    private Contact contact; 3 usages
    private Opportunity opportunity; 3 usages

    private RelationshipMapper relationshipMapper; 2 usages
    private ContactRepository contactRepository; 2 usages
    private OpportunityRepository opportunityRepository; 2 usages

    public MapLeadConversionChildrenRelationships(Account account, 1 usage  ⚡ emmprokak *
                                                Contact contact,
                                                Opportunity opportunity,
                                                RelationshipMapper relationshipMapper,
                                                ContactRepository contactRepository,
                                                OpportunityRepository opportunityRepository){

        this.account = account;
        this.contact = contact;
        this.opportunity = opportunity;
        this.relationshipMapper = relationshipMapper;
        this.contactRepository = contactRepository;
        this.opportunityRepository = opportunityRepository;
    }

    @Override 1 usage  ⚡ emmprokak
    public void execute() {
        relationshipMapper.mapLeadConversionChildrenRelationships(account, contact, opportunity);
        contactRepository.save(contact);
        opportunityRepository.save(opportunity);
    }
}
```

**Figure 3.41. Map Lead Conversion Children Relationships implementation**

After defining all “steps” of the Lead Conversion process as classes that implement the Command interface and encapsulate the business logic in the implementation of the execute method, we can proceed with creating the Command invoker class. Said class is responsible for grouping all commands to be executed. It acts as an intermediary between the calling environment and the Command interface implementing classes.

```

public class LeadConversionInvoker { 2 usages  👤 emmprokax
    private List<Command> commands = new ArrayList<>(); 2 usages

    public void addCommand(Command command) { 5 usages  👤 emmprokax
        commands.add(command);
    }

    public void executeCommands() { 1 usage  👤 emmprokax
        for (Command command : commands) {
            command.execute();
        }
    }
}

```

**Figure 3.42. Lead Conversion invoker implementation**

Inside the leadConversion method of the BP module, we can utilize the invoker to dynamically add the steps for the conversion and execute them all at the end.

```

LeadConversionInvoker invoker = new LeadConversionInvoker();

invoker.addCommand(new CreateAccountCommand(inputLead, acc));
invoker.addCommand(new CreateContactCommand(inputLead, con));
invoker.addCommand(new CreateOpportunityCommand(inputLead, opp));
invoker.addCommand(new MapLeadToChildrenCommand(inputLead, acc, con, opp, relationshipMapper, accountRepository));
invoker.addCommand(new MapLeadConversionChildrenRelationships(acc,
    con, opp,
    relationshipMapper,
    contactRepository,
    opportunityRepository)
);

invoker.executeCommands();

```

**Figure 3.43 Code segment from the leadConversion method of Business Process class**

The defined-at-runtime and loosely coupled implementation of the Lead Conversion algorithm allows us to support, extend, modify and maintain the implementation of this core business process with ease.

The other currently supported business process is that of Discount Calculation. An incoming request follows the architecture of Business Process Controller -> Business Process Service and reaches the BP module. We want to implement a logic of offering different discounts based on specific Account criteria. Aiming to offer different discounts that can be used interchangeably, we can make use of the Strategy design pattern. We will define the DiscountStrategy interface that will be implemented by the concrete Discount strategies that will contain the business logic.

```
public interface DiscountStrategy { 13 usages 3 implementations emmprokak
    double getDiscountPercentage(Account account); 2 usages 3 implementations emmprokak
}
```

Figure 3.44. Discount Strategy interface

```
public class IndustryDiscount implements DiscountStrategy { 3 usages emmprokak

    private final static Set<String> TARGET_INDUSTRIES = Set.of("IT", "Insurance");

    @Override 2 usages emmprokak
    public double getDiscountPercentage(Account account) {
        if (TARGET_INDUSTRIES.contains(account.getIndustry())) {
            return 0.04;
        }
        return 0.0;
    }
}
```

Figure 3.45. Industry Discount implementation

```
public class LoyaltyDiscount implements DiscountStrategy {

    @Override 2 usages emmprokak
    public double getDiscountPercentage(Account account) {
        return account.getClientRating() >= 9 ? 0.03 : 0.0;
    }
}
```

Figure 3.46. Loyalty Discount Implementation

For the calling environment to interact with our different discount strategies, we can declare the DiscountContext class, which acts as a wrapper that delegates the operation to the strategy through dependency injection and object composition.

```
public class DiscountContext { 3 usages  ⚡ emmprokax

    private DiscountStrategy strategy; 2 usages

    public void setStrategy(DiscountStrategy strategy) { 1 usage  ⚡ emmprokax
        this.strategy = strategy;
    }

    public double executeStrategy(Account account) { 1 usage  ⚡ emmprokax
        return strategy.getDiscountPercentage(account);
    }
}
```

Figure 3.47. DiscountContext class implementation

Lastly, we can implement a strategy factory to decouple the strategy selection mechanism from the calling environment.

```
@Component 2 usages  new *
public class DiscountStrategyFactory {

    public DiscountStrategy getStrategy(Account account) { 1 usage  new *
        if (StringUtil.stringsAreEqual(account.getIndustry(), "IT") ||
            StringUtil.stringsAreEqual(account.getIndustry(), "Insurance")) {
            return new IndustryDiscount();
        }

        if (account.getRevenue() < 200_000) {
            return new RevenueDiscount();
        }

        return new LoyaltyDiscount();
    }
}
```

Figure 3.48. DiscountStrategyFactory implementation

Through applying the described patterns, we promote loose coupling between the BP module and the implementation of the business logic. When the executeStrategy method is called, the executed strategy and its getDiscountPercentage implementation is dynamically set at runtime.

```

public ProcessOutputDTO clientDiscount(String accountId, double amount){ 1 usage  emmprokak *
    Optional<Account> accountOptional = accountRepository.findById(accountId);

    if(!accountOptional.isPresent()){
        throw new ResponseStatusException(
            HttpStatus.BAD_REQUEST,
            ErrorMessageUtility.getEntityNotFoundBySpecifier(
                Constants.Entity.ACCOUNT,
                Constants.Specifier.ID
            )
        );
    }

    Account account = accountOptional.get();
    DiscountContext discountContext = new DiscountContext();
    DiscountStrategy selectedDiscountStrategy = discountStrategyFactory.getStrategy(account);

    discountContext.setStrategy(selectedDiscountStrategy);
    double discountPercentage = discountContext.executeStrategy(accountOptional.get());

    return new ProcessOutputDTO(
        field: "totalAmount",
        processType: "discountProcess",
        String.valueOf(d: (1 - discountPercentage) * amount)
    );
}

```

Figure 3.49. Complete implementation of clientDiscount in the BP module

Seeing how the clientDiscount method is implemented, it is easy to modify the strategies involved and their specific business logic without the need for modifications in the BP module. Through the use of design patterns, we once again achieve loose coupling between application components and avoid hard-wiring business logic.

### 3.3.3 Non Design Pattern Implementation Presentation

In the above section we analyzed the Springboot application that made use of software design patterns to improve the overall quality of the implementation. In the present section, we will review how the parts of the implementation that utilized the patterns would look like if no design patterns were used. The rest of the implementation will be identical to the one described in 3.3.2, so we will not focus on their similarities, but their differences only.

In the 3.3.2 implementation, we identified 7 areas were design patterns or design pattern ideas were implemented in order to make a difference in the implementation quality. These are:

- Template Method pattern for data validation
- Command pattern on Lead Conversion
- Builder pattern for Object Mapper
- Strategy pattern for Delete Trigger
- Strategy pattern for Discount Calculation
- Generic Handling of bidirectional references
- Generic List Conversion

## Implementing data validation layer without Template method pattern

Without putting design patterns into the equation, the simplest way we can implement a `DataValidationProcessor` class would be to create a class that contains a data validation method for each Entity available.

```
@Component 2 usages new *
public class DataValidation {

    public void validateAccountData(Account account) throws DataValidationException { 1 usage new *
        if(StringUtil.stringIsEmptyOrNull(account.getCompanyName()) || StringUtil.stringIsEmptyOrNull(account.getVat())){
            throw new DataValidationException("Please provide Company Name and VAT");
        }
    }

    public void validateContactData(Contact contact) throws DataValidationException { 1 usage new *
        if(StringUtil.stringIsEmptyOrNull(contact.getFirstName()) || StringUtil.stringIsEmptyOrNull(contact.getLastName())){
            throw new DataValidationException("Please provide first name and last name for Contact");
        }
    }

    public void validateLeadData(Lead lead) throws DataValidationException { 1 usage new *
        if(StringUtil.stringIsEmptyOrNull(lead.getCompanyName())){
            throw new DataValidationException("Please provide Company Name for Lead");
        }
    }
}
```

Figure 4.1. DataValidation with no patterns implementation segment

```
public Account handleAccountEntry(Account source, Account target) throws DataValidationException { 2 usages ± emmprokax
    dataValidation.validateAccountData(source);
    target = ObjectMapper.mapAccountFields(source, target);
    target = relationshipMapper.mapAccountParents(source, target);

    return target;
}

public Contact handleContactEntry(Contact source, Contact target, boolean isInsert) throws DataValidationException { 2
    dataValidation.validateContactData(source);
    target = ObjectMapper.mapContactFields(source, target);
    target = relationshipMapper.mapContactParents(source, target, isInsert);

    return target;
}

public Lead handleLeadEntry(Lead source, Lead target) throws DataValidationException { 2 usages ± emmprokax *
    dataValidation.validateLeadData(source);
    target = ObjectMapper.mapLeadFields(source, target);
    // Lead has no parents
    return target;
}
```

Figure 4.2. Calling DataValidation class from Insert Update Trigger

Modifying the validation logic behind each Entity entails navigating to the `DataValidation` class and changing the appropriate method. Adding new Entities in the CRM means creating a new method in the `DataValidation` class for each new Entity. It is important to note that having methods for different Entities in a single class violates the Single Responsibility principle and increases

coupling between the validation logic and the Entities. Having to directly modify the class for changing validation logic or adding a new Entity validations decreases flexibility and can cause problems in regard to maintainability and scaling. For instance, to add a common validation step to all Entities, the engineer would have to manually update each Entity method, which can be cumbersome and error prone. Additionally, if the number of Entities increased significantly, the monolithic DataValidation class would become too large in size and refactoring would be necessary.

```

@Component 2 usages new *
public class ValidationHandler {

    private final Map<Class<?>, ValidationTemplate<?>> validators = new HashMap<>(); 8 usages

    public ValidationHandler() { new *
        validators.put(Account.class, new AccountValidationProcessor());
        validators.put(Contact.class, new ContactValidationProcessor());
        validators.put(Lead.class, new LeadValidationProcessor());
        validators.put(Task.class, new TaskValidationProcessor());
        validators.put(Opportunity.class, new OpportunityValidationProcessor());
        validators.put(VoiceCall.class, new VoiceCallValidationProcessor());
        validators.put(Case.class, new CaseValidationProcessor());
    }

    public <T> void validate(T entity) throws DataValidationException { new *
        ValidationTemplate<T> validator = (ValidationTemplate<T>) validators.get(entity.getClass());

        if (validator != null) {
            validator.beforeSaveProcessing(entity);
        }else{
            throw new IllegalArgumentException("No validator found for " + entity.getClass().getSimpleName());
        }
    }
}

```

Figure 4.3. Validation with Template Method design pattern

```

@Component 1 emmprokak
public class AccountValidationProcessor extends ValidationTemplate<Account> {
    @Override 1 emmprokak
    protected void validate(Account entry) throws DataValidationException {
        if(StringUtil.stringIsEmptyOrNull(entry.getCompanyName()) || StringUtil.stringIsEmptyOrNull(entry.getVat())){
            throw new DataValidationException("Please provide Company Name and VAT");
        }
    }
}

```

Figure 4.4. Extension of ValidationTemplate class for Account

```
@Autowired
private ValidationHandler validationHandler;

public Account handleAccountEntry(Account source, Account target) throws DataValidationException { 2 usages ± emmprok
    validationHandler.validate(source);
    target = ObjectMapper.mapAccountFields(source, target);
    target = relationshipMapper.mapAccountParents(source, target);

    return target;
}

public Contact handleContactEntry(Contact source, Contact target, boolean isInsert) throws DataValidationException {
    validationHandler.validate(source);
    target = ObjectMapper.mapContactFields(source, target);
    target = relationshipMapper.mapContactParents(source, target, isInsert);

    return target;
}

public Lead handleLeadEntry(Lead source, Lead target) throws DataValidationException { 2 usages ± emmprokak *
    validationHandler.validate(source);
    target = ObjectMapper.mapLeadFields(source, target);
    // lead has no parents
    return target;
}
```

**Figure 4.5. Calling design-pattern ValidationHandler from Insert Update Trigger**

With the application of the Template Method pattern, are able to tackle this issues and improve implementation quality. Centralized logic can easily be implemented in the supertype of all ValidationProcessor classes, which adhere to the Single Responsible Principle and refer to a single Entity each. If an Entity requires custom data validation logic, a new class can be created to extend the ValidationTemplate class and provide the appropriate implementation, without the need to modify existing classes that include validation logic.



## Implementing the Lead Conversion algorithm without Command pattern.

Ignoring design patterns, we can implement Lead Conversion by performing all the algorithm steps in the proper order.

```

@Transactional 1 usage 2 emmprakak
public List<EntityDTO> leadConversion(String leadId) {

    Optional<Lead> leadOptional = leadRepository.findById(leadId);

    if(!leadOptional.isPresent()){
        throw new ResponseStatusException(
            HttpStatus.BAD_REQUEST,
            ErrorMessageUtility.getEntityNotFoundBySpecifier(
                Constants.Entity.LEAD,
                Constants.Specifier.ID
            )
        );
    }

    Lead inputLead = leadOptional.get();

    Account acc = new Account();
    acc = ObjectMapper.mapLeadToAccount(inputLead, acc);
    Contact con = new Contact();
    con = ObjectMapper.mapLeadToContact(inputLead, con);
    Opportunity opp = new Opportunity();
    opp = ObjectMapper.mapLeadToOpportunity(inputLead, opp);

    relationshipMapper.mapLeadToChildren(acc, con, opp, inputLead);

    acc = accountRepository.save(acc);

    relationshipMapper.mapLeadConversionChildrenRelationships(acc, con, opp);

    contactRepository.save(con);
    opportunityRepository.save(opp);

    return Arrays.asList(acc.toDTOSimple(), con.toDTOSimple(), opp.toDTOSimple());
}

```

**Figure 4.6. No design pattern implementation of Lead Conversion algorithm**

The first thing we notice between the two implementations is that the non design pattern version contains the entire implementation of the algorithm steps. In this manner, modifications in the lead conversion algorithm steps will require changes directly in the leadConversion method of the BP module. This can lead to reduced maintainability, tight coupling and less modularity. Altering or extending the algorithm logic sets the stage for a monolithic and complex method, with a negative impact on maintainability. Additionally, the leadConversion method does a lot of things by itself, such as creating the new entries, mapping the entry fields, mapping their relationships with the

parent lead and mapping the relationships among them. This can be viewed as a deviation from the Single Responsibility principle.

```

@Transactional 1 usage  ± emmprokax
public List<EntityDTO> leadConversion(String leadId) {

    Optional<Lead> leadOptional = leadRepository.findById(leadId);

    if(!leadOptional.isPresent()){
        throw new RuntimeException(
            HttpStatus.BAD_REQUEST,
            ErrorMessageUtility.getEntityNotFoundBySpecifier(
                Constants.Entity.LEAD,
                Constants.Specifier.ID
            )
        );
    }

    Lead inputLead = leadOptional.get();
    Account acc = new Account();
    Contact con = new Contact();
    Opportunity opp = new Opportunity();

    LeadConversionInvoker invoker = new LeadConversionInvoker();

    invoker.addCommand(new CreateAccountCommand(inputLead, acc));
    invoker.addCommand(new CreateContactCommand(inputLead, con));
    invoker.addCommand(new CreateOpportunityCommand(inputLead, opp));
    invoker.addCommand(new MapLeadToChildrenCommand(inputLead, acc, con, opp, relationshipMapper, accountRepository));
    invoker.addCommand(new MapLeadConversionChildrenRelationships(acc,
        con, opp,
        relationshipMapper,
        contactRepository,
        opportunityRepository)
    );

    invoker.executeCommands();

    return Arrays.asList(acc.toDTOSimple(), con.toDTOSimple(), opp.toDTOSimple());
}

```

**Figure 4.7. Command pattern implementation of Lead Conversion algorithm**

On the other hand, the command pattern version abstracts away the implementation logic of each algorithm step. Each command is a self-contained logic unit and modifications in the step implementation are confined in the respective command class. In this way, we avoid hard-wiring and coupling the BP module with the specific business logic of the lead conversion steps. Adding new steps can be easily done by creating a new class and implementing the Command interface. New steps will only require one line of code in the leadConversion method (adding the new command to the invoker underlying command arraylist). The Command pattern architecture allows to easily maintain, extend, and modify the behaviour of the algorithm, while promoting loose coupling and the principle of Single Responsibility.

### Implementing the Object Mapper without the Builder pattern

In the design pattern version of the Java back-end, we made use of the Builder design pattern to construct the entry for insertion / updating in the Object Mapper of the Insert Update Trigger module. If we skipped the design pattern application, a simple way to accomplish the same functionality would be to manually create an object and populate its attributes with the use of setter methods.

```
public static Lead mapLeadFields(Lead source, Lead target){ 1 usage  ± emmprokax *
    Lead result = target;

    result.setCompanyAddress(source.getCompanyAddress());
    result.setCompanyIndustry(source.getCompanyIndustry());
    result.setCompanyName(source.getCompanyName());
    result.setContactEmail(source.getContactEmail());
    result.setContactMobile(source.getContactMobile());
    result.setContactPhone(source.getContactPhone());
    result.setContactRole(source.getContactRole());
    result.setContactPrefix(source.getContactPrefix());
    result.setStatus(source.getStatus());
    result.setContactPerson(source.getContactPerson());

    return result;
}

public static Opportunity mapOpportunityFields(Opportunity source, Opportunity target){ 1 usage  ± emmprokax *
    Opportunity result = target;

    result.setStatus(source.getStatus());
    result.setTitle(source.getTitle());
    result.setExpectedRevenue(source.getExpectedRevenue());
    result.setComments(source.getComments());
    result.setType(source.getType());
    result.setDescription(source.getDescription());

    return result;
}

public static Task mapTaskFields(Task source, Task target){ 1 usage  ± emmprokax *
    Task result = target;

    result.setStatus(source.getStatus());
    result.setDueDate(source.getDueDate());
    result.setName(source.getName());
    result.setReason(source.getReason());
    result.setStatus(source.getStatus());
    result.setType(source.getType());

    return result;
}
```

Figure 4.8. No design pattern setter method implementation

By explicitly setting each field through setter methods, the mapping method and the object structure become coupled. Future changes in the Entity class will require direct intervention in the mapping method. More attributes and more complex construction logic can negatively affect the mapping function by increasing its complexity.

```

public static Lead mapLeadFields(Lead source, Lead target){ 1 usage  ± emmprokak
    return new LeadBuilder(target)
        .setCompanyAddress(source.getCompanyAddress())
        .setCompanyIndustry(source.getCompanyIndustry())
        .setCompanyName(source.getCompanyName())
        .setContactEmail(source.getContactEmail())
        .setContactMobile(source.getContactMobile())
        .setContactPhone(source.getContactPhone())
        .setContactRole(source.getContactRole())
        .setContactPrefix(source.getContactPrefix())
        .setStatus(source.getStatus())
        .setContactPerson(source.getContactPerson())
        .build();
}

public static Opportunity mapOpportunityFields(Opportunity source, Opportunity target){ 1 usage  ± emmprokak
    return new OpportunityBuilder(target)
        .setStatus(source.getStatus())
        .setTitle(source.getTitle())
        .setExpectedRevenue(source.getExpectedRevenue())
        .setComments(source.getComments())
        .setType(source.getType())
        .setDescription(source.getDescription())
        .build();
}

public static Task mapTaskFields(Task source, Task target){ 1 usage  ± emmprokak
    return new TaskBuilder(target)
        .setStatus(source.getStatus())
        .setDueDate(source.getDueDate())
        .setName(source.getName())
        .setReason(source.getReason())
        .setStatus(source.getStatus())
        .setType(source.getType())
        .build();
}

```

**Figure 4.9. Builder design pattern implementation**

On the contrary, the application of the Builder pattern allows for better scaling of the object construction process. Conditional logic, null checks and default values can easily be implemented inside the concrete Builder class, without cluttering the calling environment. Direct mapping of the fields with setter methods might be fine for simple projects, but in the case of a CRM application that is expected to expand and evolve with the business requirements of its users, the flexibility, maintainability and scalability the Builder pattern offers make it the more appealing choice.

### Implementing the Delete Trigger without Strategy pattern

Without using any design patterns, we can implement the DeleteTrigger as a class that provides a function for each entity. Every entity function will make calls to a helper class that will house all handleEntityParents and handleEntityChildren methods. As a result, our no-design-pattern Delete Trigger implementation will consist of two classes, one that will group parent-child reference modifications and is exposed to the calling environment (DeleteTrigger) and one that implements the parent and children reference handling for each Entity (DeleteTriggerHelper).

```
@Component 14 usages emmprokak *
public class DeleteTrigger {

    @Autowired
    DeleteTriggerHelper deleteTriggerHelper;

    public Account handleAccountDelete(Account account){ 1 usage new *
        account = deleteTriggerHelper.handleAccountParentReferences(account);
        account = deleteTriggerHelper.handleAccountChildrenReferences(account);
        return account;
    }

    public Case handleCaseDelete(Case caseEntry){ 1 usage new *
        caseEntry = deleteTriggerHelper.handleCaseParentReferences(caseEntry);
        caseEntry = deleteTriggerHelper.handleCaseChildrenReferences(caseEntry);
        return caseEntry;
    }

    public Contact handleContactDelete(Contact contact){ 1 usage new *
        contact = deleteTriggerHelper.handleContactParentReferences(contact);
        contact = deleteTriggerHelper.handleContactChildrenReferences(contact);
        return contact;
    }

    public Task handleTaskDelete(Task task){ 1 usage new *
        task = deleteTriggerHelper.handleTaskParentReferences(task);
        task = deleteTriggerHelper.handleTaskChildrenReferences(task);
        return task;
    }
}
```

Figure 4.10. Segment of no-design-patterns DeleteTrigger

```

public Task handleTaskChildrenReferences(Task task){ 1 usage new *
    return task;
}

public Opportunity handleOpportunityParentReferences(Opportunity opportunity){ 1 usage new *
    if(!StringUtil.stringIsEmptyOrNull(opportunity.getRelatedAccountId())){
        Optional<Account> accountOptional = accountRepository.findById(opportunity.getRelatedAccountId());

        if(!accountOptional.isPresent()){
            throw new ResponseStatusException(
                HttpStatus.BAD_REQUEST,
                ErrorMessageUtility.getEntityNotFoundBySpecifier(
                    Constants.Entity.ACCOUNT,
                    Constants.Specifier.ID
                )
            );
        }

        accountOptional.get().getOpportunities().remove(opportunity);
        accountRepository.save(accountOptional.get());
    }

    return opportunity;
}

public Opportunity handleOpportunityChildrenReferences(Opportunity opportunity){ 1 usage new *

    List<Task> taskList = taskRepository.findByRelatedOpportunityId(opportunity.getId());
    for(Task task : taskList){
        task.setRelatedOpportunity(null);
        task.setRelatedOpportunityId(null);
        taskRepository.save(task);
    }

    return opportunity;
}

```

Figure 4.11. Segment of no-design-patterns DeleteTriggerHelper

Having a single class offer utilities for different Entities violates the Single Responsibility principle and leads to a large, monolithic class that becomes harder to maintain as the number of Entities increases. Centralizing operations for all Entities in one class can make it more cluttered and error prone, since adding or altering delete-operations will entail modifying the class directly. A monolithic class is more complex to understand and maintain and can face difficulty with scaling, making refactoring necessary for the future evolution of the software. In addition, this structure violates the Open & Closed principle, since modifications cannot be made by just extending the existing implementation, but they must be made by changing the original class directly.

```

public interface DeletionHandler<T> { 20 usages 7 implementations 2 emmprok
    public T delete(T entity); 7 implementations 2 emmprok
    public T handleParentReferences(T entity); 7 usages 7 implementations 2 emmprok
    public T handleChildReferences(T entity); 7 usages 7 implementations 2 emmprok
}

```

Figure 4.12. Strategy Pattern DeletionHandler Interface

```

@Override  ± emmprokak
public Case delete(Case caseEntry) {
    caseEntry = handleParentReferences(caseEntry);
    caseEntry = handleChildReferences(caseEntry);
    return caseEntry;
}

@Override  7 usages  ± emmprokak
public Case handleParentReferences(Case caseEntry){
    if(!StringUtil.isEmptyOrNull(caseEntry.getRelatedAccountId())){
        Optional<Account> accountOptional = accountRepository.findById(caseEntry.getRelatedAccountId());

        if(!accountOptional.isPresent()){
            throw new ResponseStatusException(
                HttpStatus.BAD_REQUEST,
                ErrorMessageUtility.getEntityNotFoundBySpecifier(
                    Constants.Entity.ACCOUNT,
                    Constants.Specifier.ID
                )
            );
        }

        accountOptional.get().getCases().remove(caseEntry);
        accountRepository.save(accountOptional.get());
    }

    if(!StringUtil.isEmptyOrNull(caseEntry.getRelatedContactId())){

```

Figure 4.13. Strategy Pattern DeletionHandler Implementation Segment for Case

```
@Component 14 usages  ⚡ emmprokak
public class DeleteTrigger {

    private final Map<Class<?>, DeletionHandler<?>> handlers; 2 usages

    @Autowired  ⚡ emmprokak
    public DeleteTrigger(Map<Class<?>, DeletionHandler<?>> handlers) {
        this.handlers = handlers;
    }

    @SuppressWarnings("unchecked") 7 usages  ⚡ emmprokak
    public <T> void handleReferenceDeletion(T entity) {
        DeletionHandler<T> handler = (DeletionHandler<T>) handlers.get(entity.getClass());
        if (handler != null) {
            handler.delete(entity);
        } else {
            throw new IllegalArgumentException(
                "No handler found for Entity: " + entity.getClass().getName()
            );
        }
    }
}
```

Figure 4.14. Strategy Pattern Generic Delete Trigger

In contrast to the non-design pattern implementation, with the Strategy pattern the deletion logic for each Entity is encapsulated in its own class which implements the DeletionHandler interface. Adding deletion logic for a new Entity can be achieved with creating a new class and implementing the DeletionHandler interface, ensuring no impact to the rest of the implementation. Modifying existing deletion logic requires modifications only to the appropriate Entity-specific class, making the codebase changes less error prone since each class focuses on a sole, focused responsibility. Flexibility and maintainability improve due to adhering to both the Single Responsibility and the Open & Closed principle. We can also note that with the Strategy implementation we achieve looser coupling between the Delete Trigger and its calling environment (Service Layer), since the shared interface implementation allows us to use an agnostic-to-the-entity method for invocation.



```

public boolean deleteContactById(String id){ 1 usage  ⚡ emmprokax *
    Optional<Contact> contactOptional = contactRepository.findById(id);

    if(!contactOptional.isPresent()){
        throw new ResponseStatusException(
            HttpStatus.BAD_REQUEST,
            ErrorMessageUtility.getEntityNotFoundBySpecifier(
                Constants.Entity.CONTACT,
                Constants.Specifier.ID
            )
        );
    }

    deleteTrigger.handleContactDelete(contactOptional.get());
    contactRepository.delete(contactOptional.get());

    return true;
}

```

Figure 4.15. Non-design pattern calling environment for Delete Trigger

```

public boolean deleteContactById(String id){ 1 usage  ⚡ emmprokax *
    Optional<Contact> contactOptional = contactRepository.findById(id);

    if(!contactOptional.isPresent()){
        throw new ResponseStatusException(
            HttpStatus.BAD_REQUEST,
            ErrorMessageUtility.getEntityNotFoundBySpecifier(
                Constants.Entity.CONTACT,
                Constants.Specifier.ID
            )
        );
    }

    deleteTrigger.handleReferenceDeletion(contactOptional.get());
    contactRepository.delete(contactOptional.get());

    return true;
}

```

Figure 4.16. Strategy pattern calling environment for Delete Trigger

## Implementing the Discount Calculation without the Strategy pattern

In the design-pattern version of the back-end implementation, we utilized the Strategy design pattern to calculate the discount for a specific customer, which is the core functionality of this business process. If we removed design patterns from our approach, we could model the business logic by a simple if-else statement that assigned the value to the discountPercentage local variable.

```
public ProcessOutputDTO clientDiscount(String accountId, double amount){ 1 usage
    Optional<Account> accountOptional = accountRepository.findById(accountId);

    if(!accountOptional.isPresent()){
        throw new ResponseStatusException(
            HttpStatus.BAD_REQUEST,
            ErrorMessageUtility.getEntityNotFoundBySpecifier(
                Constants.Entity.ACCOUNT,
                Constants.Specifier.ID
            )
        );
    }

    Account account = accountOptional.get();

    double discountPercentage = 0.0;

    if (StringUtil.stringsAreEqual(account.getIndustry(), "IT") ||
        StringUtil.stringsAreEqual(account.getIndustry(), "Insurance")) {
        discountPercentage = 0.04;
    } else if (account.getRevenue() < 200_000) {
        discountPercentage = 0.05;
    } else {
        discountPercentage = 0.03;
    }

    return new ProcessOutputDTO(
        field: "totalAmount",
        processType: "discountProcess",
        String.valueOf(d: (1 - discountPercentage) * amount)
    );
}
```

Figure 4.17. No design pattern implementation of clientDiscount

As we can see, without the use of the Strategy pattern, the business logic is hardcoded inside the BP module, requiring direct modifications in the BusinessProcess class to support changes in the business logic. This implementation violates the Open & Closed principle, since changes have to be made directly in the class and not through extending and creating new classes. It encompasses tightly coupled business logic that depends on many different components for alternate discounts. In addition, refactoring the code may be necessary in the future, because, as the complexity and the number of conditions increases, the clientDiscount method will become large, complex and hard to maintain and understand. Seeing how the method now focuses on both the calculation of the discount and the definition of the discountPercentage, one can claim

that it breaches the principle of Single Responsibility, since deciding on the `discountPercentage` value can be considered as a level of abstraction below the calculation formula. This can lead to cluttered, coupled and harder-to-maintain code. Lastly, we can make a point about the implementation lacking in terms of code reusability. Thanks to the discount logic being hardcoded inside the BP module, it cannot be reused by other parts of the application that might require access to discount calculation in the future. As a result, refactoring will be necessary in order to avoid code duplication.

```

public ProcessOutputDTO clientDiscount(String accountId, double amount){ 1 usage  2 emmprokak
    Optional<Account> accountOptional = accountRepository.findById(accountId);

    if(!accountOptional.isPresent()){
        throw new ResponseStatusException(
            HttpStatus.BAD_REQUEST,
            ErrorMessageUtility.getEntityNotFoundBySpecifier(
                Constants.Entity.ACCOUNT,
                Constants.Specifier.ID
            )
        );
    }

    Account account = accountOptional.get();
    DiscountContext discountContext = new DiscountContext();
    DiscountStrategy selectedDiscountStrategy = discountStrategyFactory.getStrategy(account);

    discountContext.setStrategy(selectedDiscountStrategy);
    double discountPercentage = discountContext.executeStrategy(accountOptional.get());

    return new ProcessOutputDTO(
        field: "totalAmount",
        processType: "discountProcess",
        String.valueOf(d: (1 - discountPercentage) * amount)
    );
}

```

**Figure 4.18. Strategy pattern implementation of clientDiscount**

Instead of directly including the business logic in flow control statements, the Strategy pattern implementation delegates the `discountPercentage` calculation to the classes implementing the `DiscountStrategy` interface. A `DiscountStrategyFactory` is also used, to provide the appropriate Strategy based on the use-case. Modifications can be made by creating or altering the Strategy-interface-implementing classes without touching the BP module and, thus, adhering the Open & Closed principle. The application can scale with new business rules more easily, due to loose coupling and the delegation of responsibilities out of the `clientDiscount` method. The Single Responsibility of the `clientDiscount` method is respected and the `discountPercentage` calculation is encapsulated inside the Strategy classes. Lastly, Discount Strategies can be reused outside of the BP module, setting the stage for reusable code across the entire CRM application, wherever calculating discounts might be needed.

### Implementing Handling of bidirectional references without design pattern ideas

In the 3.3.2 implementation, we presented the `handleParentChildRelationship` method which can dynamically handle the bidirectional references updates on all CRM Entities. The method doesn't make use of a specific design patterns per se, but combines influences and "good ideas" from

many design patterns to produce flexible and reusable code. It utilizes generics, bounded types, class type keys and interfaces to be able to manage complex bidirectional references between Entities with varying relationships.

Before we dive into how the `handleParentChildRelationship` was inspired by design pattern practices, we'll demonstrate how the implementation would look like without those ideas.

Since we will have no common interface for interacting with the Entity relationships and each Entity has different parent and children relationships, we have to handle the bidirectional references of each Entity separately. The `RelationshipMapperHelper` class will contain a method for each Entity-Parent Entity pair, that handles the bidirectional reference updates of a record and its parent.

```

@Transactional 1 usage new *
public Case handleCaseParentAccount(Case reqCase, Case caseToBeUpdated, Boolean isInsert) {
    String newParentAccountId = reqCase.getRelatedAccountId();
    String oldParentAccountId = caseToBeUpdated.getRelatedAccountId();

    if (!StringUtil.stringIsEmptyOrNull(oldParentAccountId) && !StringUtil.stringsAreEqual(oldParentAccountId, newParentAccountId)) {
        Optional<Account> oldParentAccountOptional = accountRepository.findById(oldParentAccountId);
        if (oldParentAccountOptional.isPresent()) {
            Account oldParentAccount = oldParentAccountOptional.get();
            oldParentAccount.getCases().remove(caseToBeUpdated);
            accountRepository.save(oldParentAccount);
        }
    }

    if (StringUtil.stringIsEmptyOrNull(newParentAccountId)) {
        caseToBeUpdated.setRelatedAccountId(null);
        caseToBeUpdated.setRelatedAccount(null);

        return caseRepository.save(caseToBeUpdated);
    }

    Optional<Account> newParentAccountOptional = accountRepository.findById(newParentAccountId);
    if (!newParentAccountOptional.isPresent()) {
        throw new RuntimeException(
            HttpStatus.BAD_REQUEST,
            ErrorMessageUtility.getEntityNotFoundBySpecifier(
                Constants.Entity.ACCOUNT,
                Constants.Specifier.ID
            )
        );
    }

    Account parentAccountFound = newParentAccountOptional.get();
    caseToBeUpdated.setRelatedAccountId(parentAccountFound.getId());
    caseToBeUpdated.setRelatedAccount(parentAccountFound);

    if (!parentAccountFound.getCases().contains(caseToBeUpdated)) {
        parentAccountFound.getCases().add(caseToBeUpdated);
    }

    if(isInsert){
        caseRepository.save(caseToBeUpdated);
    }

    accountRepository.save(parentAccountFound);
    return caseRepository.save(caseToBeUpdated);
}

```

Figure 4.19. Case with parent Account bidirectional references handling method

Methods like `getRelatedCaseId`, `getRelatedAccountId`, `getCalls`, `getCases` are specific to certain Entities and are the reason behind the many functions needed to handle all the bidirectional references.

```

@Transactional 1 usage new *
public VoiceCall handleVoiceCallParentCase(VoiceCall reqVoiceCall, VoiceCall voiceCallToBeUpdated, Boolean isInsert) {
    String newParentCaseId = reqVoiceCall.getRelatedCaseId();
    String oldParentCaseId = voiceCallToBeUpdated.getRelatedCaseId();

    if (!StringUtil.stringIsEmptyOrNull(oldParentCaseId) && !StringUtil.stringsAreEqual(oldParentCaseId, newParentCaseId)) {
        Optional<Case> oldParentCaseOptional = caseRepository.findById(oldParentCaseId);
        if (oldParentCaseOptional.isPresent()) {
            Case oldParentCase = oldParentCaseOptional.get();
            oldParentCase.getCalls().remove(voiceCallToBeUpdated);
            caseRepository.save(oldParentCase);
        }
    }

    if (StringUtil.stringIsEmptyOrNull(newParentCaseId)) {
        voiceCallToBeUpdated.setRelatedCaseId(null);
        voiceCallToBeUpdated.setRelatedCase(null);

        return voiceCallRepository.save(voiceCallToBeUpdated);
    }

    Optional<Case> newParentCaseOptional = caseRepository.findById(newParentCaseId);
    if (!newParentCaseOptional.isPresent()) {
        throw new RuntimeException(
            HttpStatus.BAD_REQUEST,
            ErrorMessageUtility.getEntityNotFoundBySpecifier(
                Constants.Entity.ACCOUNT,
                Constants.Specifier.ID
            )
        );
    }

    Case parentCaseFound = newParentCaseOptional.get();
    voiceCallToBeUpdated.setRelatedCaseId(parentCaseFound.getId());
    voiceCallToBeUpdated.setRelatedCase(parentCaseFound);

    if (!parentCaseFound.getCalls().contains(voiceCallToBeUpdated)) {
        parentCaseFound.getCalls().add(voiceCallToBeUpdated);
    }

    if(isInsert){
        voiceCallRepository.save(voiceCallToBeUpdated);
    }

    caseRepository.save(parentCaseFound);
    return voiceCallRepository.save(voiceCallToBeUpdated);
}

```

**Figure 4.20. VoiceCall with parent Case bidirectional references handling method**

It is clear that this approach is worse than the one utilizing design pattern practices. Firstly, adding a new Entity into the CRM entails creating multiple new methods on the RelationshipHandlerHelper class. A new method is required for every relationship with a parent or a child record of the new Entity. Secondly, all bidirectional reference handling methods include duplicated logic, which causes problems with maintenance and error-proness. For example, a modification in the reference handling logic would have to be replicated across every single method of the class. Additionally, we can see the Single Responsibility principle not being respected, since the RelationHandlerHelper class will offer business logic for all CRM Entities, utilizing duplicate and inflexible code.

```

@Transactional 9 usages  ⤴ emmprokak
public <P extends ParentEntity, C extends ChildEntity> C handleParentChildRelationship(
    C reqChild,
    C childToBeUpdated,
    JpaRepository<P, String> parentRepository,
    JpaRepository<C, String> childRepository,
    Class<P> parentType,
    Class<C> childType,
    Boolean isInsert
) {
    String newParentId = reqChild.getParentId(parentType);
    String oldParentId = childToBeUpdated.getParentId(parentType);

    if (oldParentId != null && !StringUtil.stringsAreEqual(oldParentId, newParentId)) {
        Optional<P> oldParentOptional = parentRepository.findById(oldParentId);
        if (oldParentOptional.isPresent()) {
            P oldParent = oldParentOptional.get();
            oldParent.removeChild(childType, childToBeUpdated);
            parentRepository.save(oldParent);
        }
    }

    if (newParentId == null) {
        childToBeUpdated.setParent(parentType, parent: null);
        return childRepository.save(childToBeUpdated);
    }

    Optional<P> newParentOptional = parentRepository.findById(newParentId);
    if (!newParentOptional.isPresent()) {
        throw new ResponseStatusException(
            HttpStatus.BAD_REQUEST,
            "Parent entity not found"
        );
    }

    P newParent = newParentOptional.get();
    childToBeUpdated.setParent(parentType, newParent);

    if (!newParent.getChildrenEntities(parentType).contains(childToBeUpdated)) {
        newParent.addChild(childType, childToBeUpdated);
    }

    if (isInsert) {
        childRepository.save(childToBeUpdated);
    }

    parentRepository.save(newParent);
    return childRepository.save(childToBeUpdated);
}

```

Figure 4.21. Generic Implementation of bidirectional reference handling

With the generic implementation all of the above mentioned issues get addressed. The result is a dynamic method that will accommodate all existing and future MNS CRM Entities and relationships. It provides better scaling while reducing a class with 500 lines of code down to 100 lines.

Even though we haven't implemented the Template Method pattern directly, the `handleParentChildRelationship` method works like a "template" by defining the bidirectional reference handling algorithm steps and deferring specifics to the implementations of the supertype function parameters. The steps of the algorithm are shared between different entities and skipping a step or altering the order of steps is not possible with the current implementation. In contrast to the general structure, which is rigid, the specifics are handled by the implementations of methods defined by the abstract parameter types, such as what children / parents to get (`ParentEntity` & `ChildEntity` method implementations of `Entities`), or how to find and update a record (JPA repositories).

As with many design patterns, interfaces are utilized as polymorphic supertype references to objects used in the algorithm logic. In order to produce generic, reusable components, interfaces are used to decouple the business logic from the specific implementation of objects and instead focus only on their abstractions (interfaces and their defined methods).

What is more, the use of Java Generics is prevalent across the `handleParentChildRelationship` method. With Generics, we can achieve great flexibility and reusability, allowing the code to operate on instances with different data-types. The data-type of the object is not known at compile time and is only defined dynamically at runtime. We can make use of bounded-types to ensure our generic parameter provides an implementation for a specific method and use that method in our generic-oriented logic implementation.

The resulting product is a generic, reusable, flexible and scalable implementation that can handle bidirectional references updates for all existing and future Entities of MNS CRM.

### **Implementing List DTO Conversion without design pattern ideas**

The `ListConverter` in the 3.3.2. implementation is drawing inspiration from design pattern ideas and practices. It utilizes interfaces and generics to provide a single function that can convert a list of any Entity records to a list of their DTO counterpart.

If we ignore the use of generics and interfaces for using polymorphic references to objects, we would have to keep a separated method for the conversion of each Entity. This means that whenever we add a new Entity to the CRM, we would have to create a new conversion method for that specific Entity in the `ListConverter` class. A modification on the conversion logic, which could for example be a new fourth DTO type, would have to be implemented across all methods of the class due to the duplicated – not reusable – code. This has negative repercussions for maintainability because of the duplicated code is inflexible and can lead to a complex, monolithic class.

Rather than following through with this implementation, we can make use of generics, bounded types and interfaces to create a generic function that is agnostic to the Entity data-type and can accommodate all CRM Entities. All the MNS CRM Entities implement the "Sendable" interface, that defines functions for converting an Entity class to its DTO counterpart. We can utilize this to ensure that all entities that function as the method input implement the `Sendable` interface and, thus, provide implementations for the DTO conversion methods. We can generically go through each Entity, convert it to DTO and add it to the resulting list, that is also the return type of the method.

```
public static <E extends Sendable<D>, D> List<D> convertEntitiesToDTOList(List<E> entities, int conversionType) {
    List<D> dtoList = new ArrayList<>();

    for (E entity : entities) {
        D dto;
        switch (conversionType) {
            case Constants.DTO.CONVERT_TO_DTO_MINIMAL:
                dto = entity.toDTOMinimal();
                break;
            case Constants.DTO.CONVERT_TO_DTO_SIMPLE:
                dto = entity.toDTOSimple();
                break;
            case Constants.DTO.CONVERT_TO_DTO_COMPLETE:
                dto = entity.toDTOComplete();
                break;
            default:
                throw new IllegalArgumentException("Invalid conversion type");
        }
        dtoList.add(dto);
    }

    return dtoList;
}
```

Figure 4.22. Generic implementation of List converter

With this generic implementation, we are using interfaces to define structure, while deferring the implementation to subtype concretes. Even though we don't directly use design patterns, we draw inspiration from their ideas in order to provide a generic function that can accommodate all existing and future CRM Entities. By not having to create other functions for list conversion operations and having to modify just a single method in order to extend functionality, we achieve flexibility, maintainability and scalability for the whole implementation.

## 4. Benchmarking

### 4.1 Establishing a framework for Benchmarking Design Patterns

In this section we will define the different metrics that will be used to quantify the effects of applying design patterns to promote software quality. Our goal is to utilize Software Quality Metrics (SQMs) to evaluate the impact of design patterns on flexibility, maintainability and extendibility in a quantitative manner.

Both in sections 3.3.2 and 3.3.3, we analyzed how the application of design patterns promoted loose coupling, reduced code duplication, allowed for modifications in accordance with the Open & Closed principle and the realized the Single Responsibility principle. We provided examples on how the software could potentially be extended in both scenarios – with the presence of design patterns or not - and we practically displayed the positive consequences they bring.

However, in this present section, we will focus on utilizing documented SQMs to benchmark the two back-end implementations and see how they compare in regard to these metrics. A custom java metrics project will be created, tasked with the goal of traversing the MNS CRM project directories and parsing each java source file. The Javaparser library will be used to handle the tokens of the java code and properly analyze them. Our custom benchmarking program will calculate a set of metrics for each back-end application and, after running the metrics-collector for both implementations, we will focus on discussing the results.



For the benchmarking process, we will use a total of 7 metrics, these being:

- Number of classes (NOC)
- Lines of Code (LOC)
- Halstead Volume (HV)
- Cyclomatic Complexity (CC)
- Maintainability Index (MI)
- Coupling Between Objects (CBO)
- Lack of Coherence in Methods (LCOM)

The Number of Classes (NOC) counts the number of classes in the source files of the application. It can be used to estimate the size of a system and, therefore, its complexity. Lines of Code (LOC) also target the estimation of a program's size, while ignoring blank lines and comments. The latter is one of the oldest metrics still used today, as it was introduced back in the 1970s (Molnar, 2020). For evaluating a system's complexity, we can also use the Halstead Volume (HV), one of the metrics introduced by Maurice Halstead in 1977. HV quantifies the effort needed to understand the software by dealing with the total and unique number of operators and operands of the codebase. The formula is defined as the sum of the total number of operators and operands times the base 2 logarithm of the sum of the number of unique operators and operands.

$$HV = (N1+N2) \times \log_2(n1+n2)$$

The metric of Cyclomatic Complexity (CC) was introduced by McCabe in 1976 and is influenced by ideas from Graph Theory. It refers to the number of independent paths through a codebase and counts the number of decision elements in a program, adding 1 plus the number of conditions and the number of decisions (Singh et al, 2011; Lee, 2014). Essentially, CC quantifies the complexity of a system's control flow.

The Maintainability Index (MI) was introduced in the early 1990s as a composite metric for quantifying software complexity (Oman & Hagemester, 1991). MI makes use of other metrics like LOC, CC and HV to estimate an applications complexity. The calculation formula includes adjusted weights to properly reflect the complexity of the software. The higher the value of the MI, the more maintainable the software is.

$$MI = 171 - 5.2 \times \log_2(HV) - 0.23 \times CC - 16.2 \times \log_2(LOC)$$

Coupling Between Objects (CBO) and Lack of Coherence in Methods (LCOM) were introduced by Chidamber and Kemerer (1994) as parts of their CK SQM suite. CBO refers to the number of coupled classes of a class which is valid when access to another class's method and members takes place. Tight coupling between modules can appear problematic in terms of maintenance and flexibility, due to having to make modifications in one part of a system because of changes elsewhere. Lower values of CBO indicate a more loosely coupled system. LCOM indicates the dissimilarity between methods based on the attributes accessed by said methods. It can help establish if a class adheres to the Single Responsibility principle by determining how many of its methods access the same members. A lower LCOM value signals more cohesive classes.

Metrics like NOC, LOC, HV and CC will be summed for the whole application, while MI, CBO and LCOM will be calculated using averages.

To calculate the above mentioned metrics for our Springboot applications, we will utilize a custom metrics java program. We will make use of recursive traversal of project directories to retrieve all the source files and parse them.

```

public class DirectoryTraversal { 1 usage

    public static void findJavaFiles(File directory, List<File> javaFiles) {
        File[] files = directory.listFiles();

        if (files != null) {
            for (File file : files) {
                if (file.isDirectory()) {
                    findJavaFiles(file, javaFiles);
                } else if (file.getName().endsWith(".java")) {
                    javaFiles.add(file);
                }
            }
        }
    }
}

```

Figure 5.1. Recursive Directory Traversal

Once all source files have been collected, we can proceed with parsing each java file separately. Metrics will also be aggregated / averaged and displayed for the complete implementation at the end of execution.

```

public static void main(String[] args) {
    MaintainabilityMetricsPerClass calculator = new MaintainabilityMetricsPerClass();
    List<File> javaFiles = new ArrayList<>();

    boolean goodImplementation = false;

    String directoryPath = goodImplementation ? "/Users/mns/IdeaProjects/mns-crm00/src/main" :
        "/Users/mns/IdeaProjects/mns-crm-backend-01/src/main";

    DirectoryTraversal.findJavaFiles(new File(directoryPath), javaFiles);

    for (File javaFile : javaFiles) {
        try {
            calculator.calculateMetricsForFile(javaFile);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    System.out.println("Total number of classes: " + calculator.totalClasses);
    System.out.println("Total lines of code: " + calculator.totalLinesOfCode);
    System.out.println("Total Cyclomatic Complexity: " + calculator.totalCyclomaticComplexity);
    System.out.printf("Total Halstead Volume: %.2f \n", calculator.totalHalsteadVolume);
    System.out.printf("Average Maintainability Index: %.2f \n", (calculator.totalMaintainabilityIndex / calculator.totalClasses));
    System.out.printf("Average CBO: %.2f \n", ((double) calculator.totalCoupling / calculator.totalClasses));
    System.out.printf("Average LCOM: %.2f \n", (calculator.totalLackOfCohesion / calculator.totalClasses));
}

```

Figure 5.2. The main method traverses the proper directory and parses each java file

```

public void calculateMetricsForFile(File javaFile) throws IOException { 1 usage
    FileInputStream in = new FileInputStream(javaFile);
    JavaParser parser = new JavaParser();
    CompilationUnit cu = parser.parse(in).getResult().orElseThrow(() -> new IOException("Failed to parse file"));

    List<String> allLines = new BufferedReader(new InputStreamReader(new FileInputStream(javaFile))).lines().collect(Collectors.toList());

    int loc = (int) allLines.stream()
        .filter(line -> !line.trim().isEmpty())
        .filter(line -> !line.trim().startsWith("//"))
        .filter(line -> !line.trim().startsWith("/*"))
        .filter(line -> !line.trim().endsWith("*/"))
        .filter(line -> !line.trim().startsWith("/*"))
        .count();

    cu.findAll(ClassOrInterfaceDeclaration.class).forEach(classDefinition -> {
        totalClasses++;
        calculateMetricsForClass(classDefinition, loc);
    });
}

```

Figure 5.3. calculateMetricsForFile implementation

Lines of Code are calculated per source file and Number of Classes is defined per class declaration

The calculateMetricsForClass method is the core metric-computation point of the metrics calculation implementation. Constructors and methods of each class are parsed to compute the CC, HV, CBO, LCOM and MI.

```

private void calculateMetricsForClass(ClassOrInterfaceDeclaration classDefinition, int linesOfCode) { 1 usage
    int classCyclomaticComplexity = 0;
    int classTotalOperators = 0;
    int classTotalOperands = 0;
    Set<String> classDistinctOperators = new HashSet<>();
    Set<String> classDistinctOperands = new HashSet<>();

    for (MethodDeclaration method : classDefinition.getMethods()) {
        classCyclomaticComplexity += calculateCyclomaticComplexity(method);

        classTotalOperators += calculateHalsteadOperators(method, classDistinctOperators);
        classTotalOperands += calculateHalsteadOperands(method, classDistinctOperands);
    }

    for (ConstructorDeclaration constr : classDefinition.getConstructors()) {
        classCyclomaticComplexity += calculateCyclomaticComplexity(constr);

        classTotalOperators += calculateHalsteadOperators(constr, classDistinctOperators);
        classTotalOperands += calculateHalsteadOperands(constr, classDistinctOperands);
    }

    int cbo = calculateCBO(classDefinition);
    double lcom = calculateLCOM(classDefinition);

    double halsteadVolume = Formula.calculateHalsteadVolume(classDistinctOperators.size(), classDistinctOperands.size(), classTotalOperators,
    double maintainabilityIndex = Formula.calculateMaintainabilityIndex(halsteadVolume, classCyclomaticComplexity, linesOfCode);

    System.out.println("Class: " + classDefinition.getNameAsString() + " with LOC = " + linesOfCode);
    System.out.println("Cyclomatic Complexity: " + classCyclomaticComplexity);
    System.out.printf("Halstead Volume: %.2f \n", halsteadVolume);
    System.out.printf("Maintainability Index: %.2f \n", maintainabilityIndex);
    System.out.println("Coupling Between Objects (CBO): " + cbo);
    System.out.println("Lack of Cohesion of Methods (LCOM): " + lcom);
    System.out.println();
}

```

Figure 5.4. The calculateMetricsForClass method implementation

The MI and HV follow specific formulas for their calculation and the implementation of which is encapsulated inside the Formula class.

```
public class Formula { 2 usages

    public static double calculateMaintainabilityIndex(double halsteadVolume, int cyclomaticComplexity, int linesOfCode) {
        return 171 - 5.2 * Math.log(halsteadVolume) - 0.23 * cyclomaticComplexity - 16.2 * Math.log(LinesOfCode);
    }

    public static double calculateHalsteadVolume(int n1, int n2, int N1, int N2) { 1 usage
        if (n1 + n2 == 0) {
            return 1;
        }

        return (N1 + N2) * (Math.log(n1 + n2) / Math.log(2));
    }
}
```

**Figure 5.5. Formula class implementation**

For the calculation of CC, all statements of the class are retrieved and, through filtering-lambdas, the total of flow control and loop statements is computed.

```
private int calculateCyclomaticComplexity(Node method) { 2 usages
    return (int) method.findAll(com.github.javaparser.ast.stmt.Statement.class).stream()
        .filter(stmt -> stmt.isIfStmt() || stmt.isForStmt() || stmt.isWhileStmt() || stmt.isSwitchStmt() || stmt.isDoStmt())
        .count() + 1;
}
```

**Figure 5.6. Method for calculating CC implementation**

For HV calculation, unique and total operators and operands are calculated by maintaining Sets for the distinct nodes and Integers for the total nodes.

```
private int calculateHalsteadOperands(Node method, Set<String> distinctOperands) {
    AtomicInteger totalOperands = new AtomicInteger();

    method.findAll(NameExpr.class).forEach(expr -> {
        distinctOperands.add(expr.getNameAsString());
        totalOperands.getAndIncrement();
    });

    method.findAll(NameExpr.class).forEach(expr -> {
        distinctOperands.add(expr.getNameAsString());
        totalOperands.getAndIncrement();
    });

    method.findAll(LiteralExpr.class).forEach(expr -> {
        distinctOperands.add(expr.toString());
        totalOperands.getAndIncrement();
    });

    method.findAll(VariableDeclarator.class).forEach(decl -> {
        distinctOperands.add(decl.getNameAsString());
        totalOperands.getAndIncrement();
    });

    return totalOperands.get();
}
```

**Figure 5.7. Operands parsing logic**

It is important to note that method calls and constructor calls (either explicit or supertype constructor invocations) are considered as operators when calculating HV. Additionally, the use of an `AtomicInteger` is necessary due to lambdas being closures and demanding all variables inside to be effectively final. `AtomicIntegers` provide a mutable `int` that can be used inside lambda functions. The use of a simple `int` would result in a compile-time error.

```
private int calculateHalsteadOperators(Node method, Set<String> distinctOperators) {
    AtomicInteger totalOperators = new AtomicInteger();

    method.findAll(BinaryExpr.class).forEach(expr -> {
        distinctOperators.add(expr.getOperator().asString());
        totalOperators.getAndIncrement();
    });

    method.findAll(UnaryExpr.class).forEach(expr -> {
        distinctOperators.add(expr.getOperator().asString());
        totalOperators.getAndIncrement();
    });

    method.findAll(AssignExpr.class).forEach(expr -> {
        distinctOperators.add(expr.getOperator().asString());
        totalOperators.getAndIncrement();
    });

    method.findAll(MethodCallExpr.class).forEach(expr -> {
        distinctOperators.add(expr.getNameAsString());
        totalOperators.getAndIncrement();
    });

    method.findAll(ExplicitConstructorInvocationStmt.class).forEach(expr -> {
        distinctOperators.add(expr.toString());
        totalOperators.getAndIncrement();
    });

    method.findAll(ConditionalExpr.class).forEach(expr -> {
        distinctOperators.add("?:");
        totalOperators.getAndIncrement();
    });

    return totalOperators.get();
}
```

**Figure 5.8. Operators parsing logic**

For calculating CBO, all method calls, accessed attributes and object instantiations are processed, with their scope being added to a Set of unique Strings. The scope of a member is the class that is responsible for its existence. Following this collection process, the class itself is excluded from the Set so that only external classes are considered in the coupling calculation.

```

classDefinition.findAll(MethodDeclaration.class).forEach(method -> {
    String returnType = method.getType().asString();
    if (isClassName(returnType)) {
        coupledClasses.add(stripGenerics(returnType));
    }

    extractGenericClasses(returnType).forEach(className -> {
        if (isClassName(className)) {
            coupledClasses.add(className);
        }
    });
});

method.getParameters().forEach(parameter -> {
    String parameterType = parameter.getType().asString();
    if (isClassName(parameterType)) {
        coupledClasses.add(stripGenerics(parameterType));
    }

    extractGenericClasses(parameterType).forEach(className -> {
        if (isClassName(className)) {
            coupledClasses.add(className);
        }
    });
});
});

classDefinition.findAll(FieldDeclaration.class).forEach(field -> {
    field.getAnnotations().forEach(annotation -> {
        if (annotation.getName().asString().equals("Autowired")) {
            String fieldType = field.getElementType().asString();
            if (isClassName(fieldType)) {
                coupledClasses.add(stripGenerics(fieldType));
            }

            extractGenericClasses(fieldType).forEach(className -> {
                if (isClassName(className)) {
                    coupledClasses.add(className);
                }
            });
        }
    });
});

coupledClasses.remove(classDefinition.getNameAsString());
return coupledClasses.size();

```

Figure 5.9. CBO implementation fragment

On the `calculateLCOM` method, all fields declared in the class are collected and a Set of the fields used by each method is maintained on a List. A 2-dimensional loop traverses all possible method pairs and, for each pair where common fields are used (intersection of the Sets), the `methodPairsWithSharedFields` counter is incremented. The ratio of methods sharing fields compared to all methods of the class signifies the LCOM rating.

```

private double calculateLCOM(ClassOrInterfaceDeclaration classDefinition) { 1 usage
    List<Set<String>> fieldsUsedPerMethod = new ArrayList<>();
    Set<String> allFieldsUsed = new HashSet<>();

    classDefinition.findAll(VariableDeclarator.class).forEach(variable -> {
        allFieldsUsed.add(variable.getNameAsString());
    });

    for (MethodDeclaration method : classDefinition.getMethods()) {
        Set<String> accessedFields = new HashSet<>();
        method.findAll(NameExpr.class).forEach(nameExpr -> {
            if (allFieldsUsed.contains(nameExpr.getNameAsString())) {
                accessedFields.add(nameExpr.getNameAsString());
            }
        });
        fieldsUsedPerMethod.add(accessedFields);
    }

    int methodPairs = 0;
    int methodPairsWithSharedFields = 0;

    for (int i = 0; i < fieldsUsedPerMethod.size(); i++) {
        for (int j = i + 1; j < fieldsUsedPerMethod.size(); j++) {
            methodPairs++;

            Set<String> intersection = new HashSet<>(fieldsUsedPerMethod.get(i));
            intersection.retainAll(fieldsUsedPerMethod.get(j));

            if (intersection.isEmpty()) {
                continue;
            }

            methodPairsWithSharedFields++;
        }
    }

    return methodPairs == 0 ? 0.0 : 1 - (double) methodPairsWithSharedFields / methodPairs;
}

```

Figure 5.10. LCOM calculation logic

After defining the metrics and the tool to be used in the benchmarking, we can proceed with applying it to our two separate Springboot applications.

## 4.2 Application to MNS CRM Implementation

In the present section we will display the results of running our metrics-calculation program for our two different back-end implementations. We will demonstrate the aggregated / averaged results per codebase and analyze them. Following this, we will focus on the metrics of the modules where the implementations differ, considering that differences come from the application



or not of design patterns. We will henceforth refer to the implementation without design patterns as “A” and the implementation with design patterns will be called “B”.

On the aggregate / average metrics of the complete implementation comparison, we can see that version B of the back-end application performs better on almost all metrics.

Metric / Implementation	No-design-patterns	Design patterns	Difference	Difference in %
Number of Classes / Interfaces (NOC)	91	129	+38	41,75%
Lines of Code (LOC)	5259	5941	+682	12,96%
Halstead Volume	42.577,91	34.463,48	-8114,43	21,06%
Maintainability Index	91,15	94,44	+3,29	3,54%
Cyclomatic Complexity	960	1008	+48	4,87%
Coupling Between Objects (CBO)	4,46	3,82	-0,64	14,34%
Lack of Coherence of Methods (LCOM)	0,29	0,27	-0,02	6,89%

**Figure 6.1. Aggregate / Average comparison of the two Springboot implementations**

For NOC, we can see that implementation B includes 38 more classes than implementation A, which constitutes an increase of 41.75%. The LOC (excluding blank lines and comments) display a difference of 682 more lines in version B which corresponds to a 12,96% increase in lines of code. These increases in code and classes can possibly be attributed to the extra classes (like concrete Builders, concrete Discount Strategies, concrete Entity-specific Deletion Handlers etc.) needed for the design pattern implementations to take form.

Since the number of classes didn't increase linearly with the LOC (41,75% to 12,96%), we can expect the average length of each class to have decreased. Considering both systems provide the same functionality and business logic, we consider more classes to be a positive characteristic, since the same business logic will be spread across more, smaller and easier-to-understand classes. Metrics agree with the described explanation since the average LOC per class are 46,05 lines for implementation B compared to the 57.79 lines of code in implementation A. This translates to a 25.49% reduction in average class size.

The HV of implementation B (34.464,48) is 21,06% lower than the HV of implementation A (42.577,91). It is important to note that, even though the codebase of implementation B is larger, the HV value is lower, signaling that the code is easier to understand and digest.

CC is the only metric where implementation A seems to have the upper edge against the design pattern implementation. More specifically, CC of implementation B (1008) is 4,87% larger than implementation A's (960). This difference isn't by a lot and can possibly be explained by the additional control flow some design patterns bring. For example, both the Strategy pattern (used twice) and the Factory pattern (used once) can include decision points and control flow statements.

When it comes to CBO, implementation B demonstrates a lower coupling between classes. Implementation A has a CBO of 4,46 and implementation B has 3,82. Both applications reflect a low coupling ratio with implementation B having 14,34% looser coupling. As described in the design pattern consequence examples in 3.3.2, we initially expected the gap between the two implementations to be higher, but we can at least see that there is a difference, validated by the CK CBO metric. We will explore how the design pattern implementation sections of the codebase compare to the non-design pattern segments in more detail later.

Both implementations seem to have highly cohesive classes with the implementation B scoring slightly higher cohesion than implementation A. The former (0,27) appears 6.89% more cohesive compared to the latter (0,29). We can also note that since LCOM is calculated as a ratio of the class methods that share the same attributes compared to the total methods of the class, the

cohesion of both implementations is in fact lower than demonstrated in this present LCOM calculation. This is due to CRM Entity and DTO classes that contain a lot of getters / setters. Since each getter / setter deals only with one member, these classes rank low in cohesion. Because this influences equally both implementations' LCOM score, we do not consider it as interference with our benchmarking analysis.

Having reviewed the total / average metrics for the complete implementations, we can review how the two implementations differ in their metrics in the parts where design patterns were implemented.

### Template Method pattern for data validation

When using Template Method to offer the data validation functionality and by collecting the metrics for the individual classes involved, we can see that implementation B scores better in almost all SQMs.

Data Validation without Design patterns	LOC	HV	MI	CC	CBO	LCOM	NOC
InsertUpdateTrigger	56	908,18	68,76	7	10	0	1
DataValidation	43	612,11	73,48	14	9	1	1
Total / AVG	99	1.520,29	71,12	21	9,50	0,50	2

Figure 6.2. Implementation A Data Validation module metrics

Data Validation with Design patterns	LOC	HV	MI	CC	CBO	LCOM	NOC
InsertUpdateTrigger	58	838,93	68,6	7	10	0	1
ValidationHandler	29	154,15	89,56	3	9	0	1
ValidationTemplate	12	9,51	118,34	3	0	1	1
AccountValidationProcessor	15	39,3	107,58	2	3	0	1
ContactValidationProcessor	16	39,3	106,53	2	3	0	1
CaseValidationProcessor	15	88,76	103,34	2	3	0	1
LeadValidationProcessor	16	16,25	111,12	2	3	0	1
TaskValidationProcessor	17	63	103,1	2	3	0	1
OpportunityValidationProcessor	16	63	104,08	2	3	0	1
VoiceCallValidationProcessor	16	16,25	111,12	2	3	0	1
Total / AVG	210	1.328,45	102,34	27	4,00	0,10	10

Figure 6.3. Implementation B Data Validation module metrics

Even though the LOC are more than doubled in the case of implementation B, these lines are spread through a lot more, smaller classes. In spite of the more code, the HV value is 13,46% lower for implementation B compared to implementation A, signaling lower effort needed to understand the code.

However, we can see that CC is 25% higher for the design pattern implementation. We believe that this is because of the many classes involved with implementing the Template Method design pattern. The average CC per class is significantly smaller (2,7 for implementation B and 10,5 for implementation A), but a lot of classes make it add up in the summary. When it comes to MI, implementation B appears as 35,99% more maintainable. This large difference can be explained by the newly added design pattern classes, that each displays a high MI value and influences the total implementation average.

Lastly, implementation B performs fundamentally better regarding CBO (81,48% less coupling) and LCOM (80% more method cohesion). We can explain that performance difference by the

design pattern related classes, that display healthy CBO and LCOM values, influencing the total implementation average.

### Command pattern on Lead Conversion and Strategy, Factory for Discount Calculation

We focused our efforts in collecting data for the specific classes involved in the Business Process module in both implementations. The BusinessProcess class includes functionality for both the Lead Conversion and Client Discount processes, so we handle them as one comparison unit. The Command, Strategy and Factory patterns are utilized in this context inside implementation B.

In the comparison results we can see that the design pattern version doesn't score very well in this instance. Let's go through the specific metrics and we will discuss possible reasons at the end of the section for the present use case.

Business Processes without Design patterns	LOC	HV	MI	CC	CBO	LCOM	NOC
BusinessProcess	90	1011,74	60,74	6	18	0	1

Figure 6.4. Implementation A Lead Conversion module metrics

Business Processes with Design patterns	LOC	HV	MI	CC	CBO	LCOM	NOC
BusinessProcess	98	797,66	61,06	4	24	0	1
DiscountStrategyFactory	20	66,44	99,96	3	6	0	1
DiscountStrategy	5	1	144,7	1	1	0	1
DiscountContext	11	16	117,28	2	2	0	1
IndustryDiscount	14	20,68	112,04	2	3	0	1
LoyaltyDiscount	11	22,46	115,74	1	1	0	1
RevenueDiscount	12	18,09	115,23	2	1	0	1
LeadConversionInvoker	14	22	111,71	2	2	0	1
Command	4	1	148,31	1	0	0	1
CreateAccountCommand	16	37,15	106,83	2	1	0	1
CreateContactCommand	16	37,15	106,83	2	1	0	1
CreateOpportunityCommand	16	37,15	106,83	2	1	0	1
MapLeadToChildrenCommand	31	121,89	89,93	2	0	0	1
MapLeadConversionChildrenRelationships	36	117,29	87,71	2	0	0	1
Total / AVG	304	1.249,52	108,87	28	3,07	0,00	14

Figure 6.5. Implementation B Lead Conversion module metrics

We again notice that even though the design pattern version includes more than triple the lines of code of implementation A, these lines are separated across more smaller-sized classes. However, the size difference here is much larger than the Template Method implementation analyzed in the previous segment.

Because of the much larger implementation B, the latter displays a 23,5% higher HV value, which can be attributed to the added complexity. Implementation B still manages to achieve a better MI, which occurs due to the addition of many new classes with healthy MI stats. CC is 366.66% higher in the design pattern version and this can be attributed to implementation B having 14 times more classes and 3 times the code to achieve the same behaviour.

The difference between CBO averages can be explained by the many more classes involved in the implementation B, that lower the average metric for CBO. The BusinessProcess class itself, displays higher coupling (28,57%) in the case of design pattern application. This is because the BusinessProcess class continues to maintain references to the objects it did back in implementation A and also adds references to new objects related to the design pattern

application (e.g. LeadConversionInvoker, DiscountContext, DiscountStrategyFactory). Both implementations appear equally cohesive in terms of LCOM.

From the above analysis, we can see that the design pattern version appears to have the upper edge only in terms of MI. It seems that for this specific application of design patterns, the results aren't so clear when it comes to the quantitative metrics. This could be for a couple of reasons such as overengineering and improper application of patterns. By overengineering, we mean using overkill methods to solve a simple problem in the degree that unnecessary complexity is introduced. The improper application of patterns in this scenario refers to the design pattern classes abstracting just the process steps and not also being responsible for encapsulating the creation of the objects they use, instead relying on receiving them from the calling environment (BusinessProcess class). In an alternate scenario, we could have seen lower LOC, CBO and HV for the BusinessProcess class itself and, as a result, for the whole implementation B's Business Process module.

```

Lead inputLead = leadOptional.get();
Account acc = new Account();
Contact con = new Contact();
Opportunity opp = new Opportunity();

LeadConversionInvoker invoker = new LeadConversionInvoker();

invoker.addCommand(new CreateAccountCommand(inputLead, acc));
invoker.addCommand(new CreateContactCommand(inputLead, con));
invoker.addCommand(new CreateOpportunityCommand(inputLead, opp));
invoker.addCommand(new MapLeadToChildrenCommand(inputLead, acc, con, opp, relationshipMapper, accountRepository));
invoker.addCommand(new MapLeadConversionChildrenRelationships(acc,
    con, opp,
    relationshipMapper,
    contactRepository,
    opportunityRepository)
);

invoker.executeCommands();
    
```

**Figure 6.6. Implementation B: objects being created in BusinessProcess and passed to design pattern classes**

The positive effects we documented in 3.3.2 (Lead conversion steps are properly encapsulated, identifying the discount percentage is abstracted etc.) still apply. However the metrics tell us that, even though certain quality characteristics have been achieved, a lot of complexity was introduced and the module is a candidate for refactoring.

**Builder pattern for Object Mapper**

The Object Mapper class is responsible for setting the field values of objects to be inserted / updated. The application of Builder pattern has been used in implementation B to abstract the object instantiation process. Even though we described the benefits of this design decision back in 3.3.2, the metrics present a different picture.

Object Mapper without Design patterns	LOC	HV	MI	CC	CBO	LCOM	NOC
ObjectMapper	137	3818,38	44,96	15	11	0	1

**Figure 6.7. Implementation A Object Mapper module metrics**

Object Mapper with Design patterns	LOC	HV	MI	CC	CBO	LCOM	NOC
ObjectMapper	138	2814,18	46,66	14	17	0	1
AccountBuilder	63	313,82	70,54	15	1	0	1
ContactBuilder	67	330,34	69,04	16	2	0,15	1
LeadBuilder	56	253,32	74,02	13	1	0	1
CaseBuilder	48	195,4	78,33	11	1	0	1
OpportunityBuilder	39	140,65	83,86	9	1	0	1
TaskBuilder	36	111,13	86,61	8	1	0	1
VoiceCallBuilder	36	111,13	86,61	6	1	0	1
Total / AVG	483	4.269,97	74,46	92	3,13	0,02	8

**Figure 6.8. Implementation B Object Mapper module metrics**

Implementation B demonstrates 252,55% more code and 8 times the classes to achieve the same result. This large gap in implementation size is reflected in HV where implementation B (4269,97) has a 11.16% higher score compared to implementation A (3818,38).

The MI is better (65,61% to be exact) for the design pattern version due to the average being influenced by the new “builder” classes, where higher MI values can be found. On the contrary, CC is far worse showing a spike from 15 to 92 for implementation B. This is because each method increments CC by 1 and this applies for every builder for every field in the object-to-be-constructed. With the addition of all new builder classes and the handling of every Entity’s field, the total CC value is raised drastically. The average CBO is better for implementation B, although the ObjectMapper class itself displays higher coupling (54,54%). We can explain this by the ObjectMapper class continuing to reference the Entities directly (as function arguments and method return types), while also adding references to the new builder classes. Finally, LCOM is very slightly worse for implementation B, but this can be solely attributed to a single method of the ContactBuilder class that doesn’t make use of the respective Contact member field.

The achieved benefits of abstracting the object instantiation process from the Object Mapper that were established in 3.3.2 are still true. However, the metrics suggest that refactoring might be necessary for the Object Mapper module.

### Strategy pattern for Delete Trigger

The Strategy pattern was used to create a dynamic Deletion Trigger, that handles removing the references of the record-to-be-deleted from other Entity entries. As in the first example analyzed, the positive influence of the design patterns can be validated using the described metrics.

Delete Trigger without Design patterns	LOC	HV	MI	CC	CBO	LCOM	NOC
DeleteTrigger	44	569,97	75,09	7	8	0	1
DeleteTriggerHelper	231	3234,77	33,45	32	18	0,75	1
Total / AVG	275	3.804,74	54,27	39	13,00	0,38	2

**Figure 6.9. Implementation A Delete Trigger module**

Delete Trigger with Design patterns	LOC	HV	MI	CC	CBO	LCOM	NOC
DeleteTrigger	37	96,21	88,07	3	1	0	1
DeletionHandler	6	1	141,28	3	0	1	1
DeletionHandlerConfig	29	120,46	91,31	1	11	0	1
AccountDeletionHandler	82	808,85	63,64	5	10	0,67	1
ContactDeletionHandler	47	250,67	78,75	5	7	1	1
CaseDeletionHandler	68	449,33	69,27	7	8	1	1
LeadDeletionHandler	44	151,72	82,89	3	2	1	1
OpportunityDeletionHandler	57	389,4	73,34	5	8	1	1
TaskDeletionHandler	66	443,31	69,83	7	8	1	1
VoiceCallDeletionHandler	68	443,31	69,34	7	8	1	1
Total / AVG	504	3.154,26	82,77	46	6,30	0,77	10

Figure 6.10. Implementation B Delete Trigger module

The design pattern version uses 5 times the classes and almost double the lines of code to achieve the same functionality. It is important to note that this LOC difference between the two codebases' implementations (83,27% increase) appears healthier than the previous two examples analyzed. This is also visible in the MI, where implementation B scores better, with a 18,69% lower MI value. Additionally, even though the number of classes is larger in implementation B, the CC value hasn't risen as much, being just 17,94% higher for the design pattern version.

CBO is also a metric where implementation B performs better, showing a 69,53% difference compared to implementation A. It is also nice to see that the very high coupling (18) of implementation A's DeleteTriggerHelper is broken down in more, less coupled classes for implementation B. In this present case, the only metric where implementation B doesn't have the upper hand is LCOM, where implementation A scores significantly better. The worse LCOM score can be attributed to every Entity-specific "DeletionHandler" class having an LCOM rating of 1. This occurs because the members of these classes are not used by all their methods, since each class has a method for handling parent references and a method for handling children references. It is unlikely that an Entity will have the same Entity Type both as a parent and a child, making it impossible to encounter two methods that make use of the same "repository" member in the same DeletionHandler class.

In contrast to the two previous scenarios, the benefits of design patterns are clearly reflected in the SQMs for this module.

### Generic Handling of bidirectional references

For this and the following case, implementation B doesn't make use of design patterns directly to solve a problem, but instead takes inspiration from them and applies it in order to create reusable generic code. The metrics agree with the described positive effects in section 3.3.2 and implementation B performs better.

Non - Generic Bidirectional References	LOC	HV	MI	CC	CBO	LCOM	NOC
RelationshipMapper	67	1353,47	61,25	18	11	0	1
RelationshipHandlerHelper	397	8116,36	12,3	65	22	0,74	1
Total / AVG	464	9.469,83	36,78	83	16,50	0,37	2

Figure 6.11. Implementation A bidirectional references handling



Generic Bidirectional References	LOC	HV	MI	CC	CBO	LCOM	NOC
RelationshipMapper	155	1410,31	47,68	17	18	0	1
RelationshipHandlerHelper	77	1033,22	62,47	9	12	1	1
Total / AVG	232	2.443,53	55,08	26	15,00	0,50	2

**Figure 6.12. Implementation B bidirectional references handling**

The goal of reducing code duplication by using design pattern ideas is achieved, with implementation B having half the code of implementation A to support the same business logic. HV is greatly smaller for the former, demonstrating a drop of 74,19% from implementation A to implementation B.

MI is 49,75% higher in implementation B, with implementation A's RelationshipHandlerHelper class bad MI score improving drastically. The design pattern inspired version also manages to score slightly better in CBO. However, LCOM displays a higher value for implementation B, which is due to less cohesion for the RelationshipHandlerHelper class.

In this use case, we can see that implementation B is clearly performing better in all metrics except LCOM.

### Generic List Conversion

For list conversion, implementation B takes inspiration from design patterns to reduce code duplication and create more scalable and maintainable code. The positive impact described in section 3.3.2 is validated by the use of SQMs.

Non Generic List Conversion	LOC	HV	MI	CC	CBO	LCOM	NOC
ListConverter	113	1186,21	54,39	14	17	0	1

**Figure 6.13. Implementation A list conversion**

Generic List Conversion	LOC	HV	MI	CC	CBO	LCOM	NOC
ListConverter	31	139,81	89,22	2	4	0	1

**Figure 6.14. Implementation B list conversion**

By applying design pattern characteristics to our ListConverter utility, we are able to reduce LOC by 72,56%. The drop in HV (82.2%) is even more impressive with implementation A's HV value being 1186,21 compared to 139,81 of implementation B. The latter also scores better in terms of MI, appearing as 61,03% more maintainable. Both CC and CBO follow in the same pattern, with CC decreasing by 85,71% and CBO decreasing by 76,47%. We observe no impact in LCOM caused by the application of design pattern ideas.

## 4.3 Results and Observations

In section 4.2 we analyzed both back-end implementations' performance in regard to SQMs. We initially fixed our attention on the per implementation total / average metrics and then proceeded with the comparison of the two implementations in the parts where they differ, which are no other than the design pattern applications themselves.

Regarding the application total / average metrics, we can see implementation B appears to perform better across all metrics except CC. We attribute the higher CC to the additional logic, methods and control flow statements brought by the design pattern introduction. Even though the design pattern version appears to have a larger number of classes and more lines of code, we consider this to be a positive characteristic, since the same functionality is provided by dividing code into smaller, easier to maintain classes. The average class size has decreased, and the more lines of code are a product of introducing new classes for the design pattern application.

No matter the higher line count, the HV appears to have dropped significantly for implementation B, signaling that the latter is less complex and easier to comprehend. CC, HV and LOC contribute to the calculation of the MI metric, in which implementation B performs better. The latter also manages to succeed both in terms of CBO and LCOM, demonstrating looser coupling and more coherent classes that better adhere to the Single Responsibility principle.

It is important to note that both implementations scored well on all metrics, showing that both implementations comply with certain quality standards.

Following the total / average analysis, we analyzed the effects of design pattern application to materialize certain functionality. Across the different case studies, we identified cases where the positive impact of design patterns was visible across the metrics used in this present paper, with the exception of course of CC. In this group, we include the Template Method for Data Validation, Strategy pattern for Delete Trigger, Generic Bidirectional Reference Handling and Generic List Conversion. There were also scenarios where the design pattern version performed worse than implementation A in many metrics, as was the case with Strategy, Command pattern and Factory pattern for Business Process and Builder pattern for Object Mapper.

CBO was the metric that implementation B had the upper hand in every specific use case of design pattern introduction. For HV and MI, implementation B scored better in four out of the six use cases. In CC, implementation A had the upper hand, with implementation B only performing better in the case of Generic Bidirectional References and Generic List Conversion, which corresponds to two out of the six scenarios. Lastly, for LCOM, implementation B performed better only in the case of Template Method for Data validation, with implementation A outperforming the latter three times and tying each other twice.

## Conclusions

In this present paper, we focused on Software Design patterns and how their positive impact can be reflected in Software Quality Metrics. A CRM Springboot application was developed twice, with and without the use of design patterns, in an attempt to practically assess the impact of the latter in a quantitative manner. A set of literature - derived Software Quality metrics was defined and used to benchmark the effects of design pattern application in our two separate back-end implementations.

In application-wide metrics, the design pattern version of the CRM app performed better in all utilized SQMs, apart from Cyclomatic Complexity. This can possibly be attributed to the larger number of classes, methods and control flow statements that come with the design pattern introduction. We also compared metrics that referred to specific application modules, with one implementation making use of design patterns and the other not. We identified use-cases where design patterns had a positive impact in all metrics (except CC), but there were also scenarios where the design pattern version showed worse results in regard to our metrics suite. Possible explanations for this behavior could be faulty application of design patterns and overengineering.



A lesson that can be drawn from the above stated is that design patterns have a positive influence in software quality and structure, as long as they are used in a proper manner and don't introduce unnecessary complexity and overhead. More research will of course be required to investigate further the correlation of SQM values and design pattern application.

## Bibliography

Ampatzoglou, A., Chalarampidou, S., & Stamelos, I. (2013). Research State of the Art on GoF Design Patterns: A Mapping Study. *Journal of Systems and Software, Volume 86, Issue 7*, Pages 1945-1964.

Aratchige, R., Gunaratne, M., Kariyawasam, K., & Weerasinghe, P. (2022). An Overview of Structural Design Patterns in Object-Oriented Software Engineering. *SW Modelling CI Wk*.

Aversano, L., Cerulo, L., Canfora, G., & Di Penta, M. (2007). An Empirical Study on the Evolution of Design Patterns. *Conference Paper*, DOI: 10.1145/1287624.1287680.

B. Eckel. Thinking in Patterns with Java.

[https://archive.org/details/Bruce\\_Eckel\\_Thinking\\_in\\_Patterns\\_with\\_Java](https://archive.org/details/Bruce_Eckel_Thinking_in_Patterns_with_Java)

Chidamber, S., & Kemerer, C. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 476–493.

De Laurentis, L. (2019). From Monolithic Architecture To Microservices Architecture. *2019 IEEE International Symposium on Software Reliability Engineering Workshops*.

Drouin, N., Badri, M., & Toure, F. (2013). Analyzing Software Quality Evolution Using Metrics: An Empirical Study On Open Source Software. *Journal of Software*, 8(10).

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1993). Design Patterns: Abstraction and Reuse of Object-Oriented Design. *ECOOP '93 Conference Proceedings, Springer-Verlag Lecture Notes in Computer Science*.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass: Addison-Wesley.

Harmes, R., & Diaz, D. (2008). *Pro JavaScript Design Patterns*. USA: Apress.

Henry, S., & Kafura, D. (1984). The Evaluation of Software Systems' Structure Using Qualitative Software Metrics. *Software- Practice and Experience*, 14(6).

Lee, M. (2014). Software Quality Factors and Software Quality Metrics To Enhance Software Quality Assurance. *British Journal of Applied Science & Technology*, 4(21).

McCabe, T. (1976). A Complexity Measure. *IEEE Transactions On Software Engineering*, SE-2(4).

Mohammed, M., & Elish, M. (2013). A Comparative Literature Survey of Design Patterns Impact on Software Quality. *Conference Paper*, DOI: 10.1109/ICISA.2013.6579460.

Molnar, A. (2020). Evaluation of Software Product Quality Metrics. In Damiani, E., Spanoudakis, G., & Leszek, M. (eds) (2020). *Evaluation of Novel Approaches to Software Engineering*. Switzerland: Springer.

O'Connor, R., Elger, P., & Clarke, P. (2017). Continuous Software Engineering: A Microservices Architecture Perspective. *Wiley Journal of Software: Evolution and Process*, e1866.

Oman, P., & Hagemester, J. (1992). Metrics for Assessing a Software System's Maintainability. *Proceedings of the Conference on Software Maintenance, IEEE*, pp. 337-344.

Singh, G., Singh, V., & Singh, D. (2011). A Study of Software Metrics. *IJCEM International Journal of Computational Engineering & Management*, 11.

Tichy, W. (1998). A Catalogue of General-Purpose Software Design Patterns. *UIUC PATTERNS GROUP VERSION Karlsruhe Institute of Technology*, DOI: 10.1109/TOOLS.1997.65474.

Trivedi, P., & Kumar, R. (2012). Software Metrics To Estimate Software Quality Using Software Component Reusability. *IJCSI International Journal of Computer Science Issues*, 9(2).

Welker, K. (2001). The Software Maintainability Index Revisited. *The Journal of Defense Software Engineering*, August 2021.