

Article

Design Patterns and Electric Vehicle Charging Software

Maria Meheden¹, Andrei Musat¹, Andrei Traciu¹, Andrei Viziteu¹, Adrian Onu¹, Constantin Filote² 
and Maria Simona Răboacă^{2,3,*} 

¹ Research and Development Department, ASSIST Software, Str. Tipografiei Nr.1, 720043 Suceava, Romania; maria.meheden@assist.ro (M.M.); andrei.musat@assist.ro (A.M.); andrei.traciu@assist.ro (A.T.); andrei.viziteu@assist.ro (A.V.); adrian.onu@assist.ro (A.O.)

² Faculty of Electrical Engineering and Computer Science, Stefan Cel Mare University of Suceava, Str. Universitatii Nr.13, 720229 Suceava, Romania; filote@usm.ro

³ National Research and Development Institute for Cryogenic and Isotopic Technologies—ICSI Rm. Valcea, 240050 Ramnicu Valcea, Romania

* Correspondence: simona.raboaca@icsi.ro

Abstract: The development and maintenance of complex software systems, with ever-changing requirements has benefited from the implementation of design patterns, which ensure a higher degree of maintainability. The present article illustrates the essential role of design patterns in sketching the software architecture for an electric vehicle charging management platform. We have integrated a series of design patterns to create a solid base structure for the API (Application Programming Interface). Furthermore, we have explored cloud design patterns in the deployment process to ensure a proper multi-tenant cloud application with the best possible tenant isolation for the cost. The aim of this paper is to offer readers an introduction in the case study theme, to describe a complex platform development through design patterns. With proper examples from real projects debated in the industry. The paper demonstrates the design patterns applicability from software level to cloud resources plan and advocates for a high-quality solution in every segment of project development. In conclusion, the results are promising, and the functionality of the established methods on this type of platforms will be evaluated during the implementation period. Overall, design patterns have proven to be essential in the development process, ensuring effective team communication and the delivery of qualitative software solutions.



Citation: Meheden, M.; Musat, A.; Traciu, A.; Viziteu, A.; Onu, A.; Filote, C.; Răboacă, M.S. Design Patterns and Electric Vehicle Charging Software. *Appl. Sci.* **2021**, *11*, 140. <https://dx.doi.org/10.3390/app11010140>

Received: 2 December 2020

Accepted: 19 December 2020

Published: 25 December 2020

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



Copyright: © 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

Keywords: software development; software design pattern; cloud design patterns; multi-tenant architecture; electric vehicle charging; electric vehicle charging management platform

1. Introduction

1.1. Design Patterns in Software Development

Software development has undergone a rapid development over the last decades, the software products having a variety of purposes and being subject to the constantly evolving needs of the customer. Therefore, the ever-changing requirements make the system more intricate and the code harder to navigate, with potential snowball effects in different areas of the code, in the absence of clear separation of concerns. When designing a software product, developers must anticipate aspects that may not be noticeable in the early stages of implementation. Freshly written code is susceptible to subtle issues, which cannot be detected immediately, and can trigger major flaws. A proactive approach to software development meant to address this predicament is the implementation of design patterns, which could be viewed as an indicator of the system's long-long term survival. Furthermore, using patterns contributes to the improvement of code readability for coders, which further ensures efficient system maintenance.

Therefore, design patterns are recognized as reusable solutions to reoccurring glitches that provide improved software maintainability. However, even though there are multiple types of design patterns, which can be use in multiple scenarios, there is a gap when it

comes to practice and the implementation of patterns in specific types of applications, which have not been addressed to their full potential, so far.

Retracing the origins of the design patterns term, we learn that it originated as an architectural concept, coined by the architect Christopher Alexander in his 1977 publication entitled *A Pattern Language: Towns, Buildings, Construction*. Due to his experience, the architect observed that there are several design constructs, which can generate the envisaged results when applied to categories of recurring problems. Consequently, he set out to register these patterns, for others to benefit from his observations and design their own products. One of the most common examples with respect to patterns is “a place to wait”. This general design pattern applies both to hospital waiting rooms and bus stops and as different as these two instances may be. This is indicative of the pattern’s generality. In software engineering, solutions that have proven successful in generating a pathway in solving problems that occur in object-oriented software systems, have been handled by four authors in 1994 in the book entitled *Design Patterns: Elements of Reusable Object-Oriented Software*. A good approach in making these decisions is to plan the software solution based on the design patterns concept. It is broadly utilized by software engineers to manufacture complex systems. Design patterns have been explored by numerous specialists in the last decades. This prompts the development of complex EV (electric vehicle) platforms shaped under the design patterns umbrella.

The goal of this article is to elaborate an overview of current design patterns used in software as a service (SaaS) type of software applications, to identify appropriate design patterns and suggest improvements for a specific use case: software for EV charging stations namely an EV charging cloud app, which is a distributed application that contains multiple technologies: Backend Microservices, Frontend JavaScript Based app, Mobile (IOS, Android), Blockchain and SQL Database. It also integrates cloud resources such as storage containers, Docker containers for deployment, continuous integration/continuous delivery, etc. It is possible to achieve a cloud native application with zero down-time, with around the clock availability, automatic scaling and failure tolerance. The current article is part of the SMART EVC project, being supported by a grant of the Romanian Ministry of Education and Research, CCCDI-UEFISCDI, project number PN-III-P2-2.1-PTE-2019-0642, within PNCDI III.

The present article is structured in six parts.

The first section consists of an introduction to the subject and a short state-of-the-art review on design patterns, studies and advancements today.

The second part is an overview of existing software design patterns and their appropriate use cases according to the literature, presenting both the advantages and the potential disadvantages of their use. We also make a clear distinction between design patterns and design principles.

The next section is related to our case study, where we implemented a software concept for a EV charging platform, using the most suitable design patterns, and proposing a cloud architecture based on cloud design pattern in order to ensure a reliable software product. Moreover, for the purpose of ensuring a proper tenancy model for a multitenant application as an EV charging management platform, we analyzed the use of multitenant design patterns in the deployment process. The closing sections present the results and conclusions reached during the study.

1.2. State of the Art

For reviewing the state of the art in matters of design patterns, we studied the databases of IEEE, Web of Science and Scopus, ISI Manuscripts used the Web of Science extensions in constructing our database. We proposed to build a qualitative collection of materials filtered and selected one by one, to highlight their essence and to elaborate an overview of current design patterns used in software as a service (SaaS) type of software applications. We have chosen to make such a rigorous selection so we will be able to provide a clear and revised solution.

Flora Amato and Francesco Moscato [1] present the importance of mapping frameworks to Cloud Architecture due to the necessity of managing big data and large data sets as quickly and as reliably as possible. New frameworks, algorithms and architectures are being used to optimize and improve performance in various fields, for a wide range of users, from big companies to small ones, or public utilities. The MapReduce framework represents the base for many services on a variety of analyses. Nelio Cacho et al. [2] explore the GoF design patterns using the scalability of AOP and studying its drawbacks and benefits. The pattern composability is affected by the dominant factor, the category of the pattern, and by the programming abstractions.

Rostislav Fojtik [3] updates the requirements on teaching programming and algorithmization in the software industry, presenting the way in which design patterns can be used in teaching programming. The improvement in the teaching technique has been shown to increase the students' pee in programming.

The author describes the impact of programming languages on the education area. He performed an analysis of the main programming languages that are proposed in educational programs. In a separate section, the author suggests that programming courses should be presented around the design patterns topic with relevant examples.

In the current paper, however, we present the design pattern topic through an application that aims the Electric Vehicle domain, which is currently in the top electric engineering industry research. We will perform an example of software design pattern applicability from each category, as follows, creational, structural and behavioral.

Wouter van Diggelen and Maarten Overdijk [4] highlight the issues that may appear on account of a detachment between practice and design, in the process of designing an e-learning environment. They consider that theory development plays a crucial role in development of design patterns. They rely on a systematic examination of the encountered issues and on the establishment of conceptual models meant to determine the definition and evaluation of design patterns. The research presents the way the proposed approach has been applied in a classroom, for developing design patterns for networked learning.

Shahid Hussain, Jacky Keung and Arif Ali Khan [5] propose a method to automatically select and classify design patterns for software developers. The approached technique uses unsupervised learning Fuzzy c-means. They also introduce Ensemble-IG, a new feature selection method, meant to overcome the drawbacks of the proposed approach. To assess the effectiveness of the method they used an evaluation model on the design pattern collection.

In the cited work, the authors exposed a method to classify the design patterns in order to help developers to identify the suitable pattern to a specific issue. We consider this subject an interesting approach. However, we maintained the value of study and comprehended the design patterns concept and found correspondence in real problem solving issues. With the aim to illustrate this technique, we provide examples that are composed of an introduction, an issue that can be solved and the implementation in the EV platform development.

Davide Arcelli and Daniele Di Pompeo [6] present a new path that applies design patterns, centered on the refactoring software artifacts with the purpose of eliminating the performance antipatterns. The research presents future research directions and the results the applied method obtained.

In this previous paper, the authors highlight the significance of design patterns achievement in the software solutions development. We agreed that design pattern should represent the tone in a refactoring process. Moreover, we proposed design patterns as a cornerstone in the platform architecture plan.

Neven A. M. ElSayed et al. [7] present the blended model view controller, which enables common user interface controls to function alongside printed information by using a semantic matrix. The combination of augmented reality and design pattern include details-on-demand, menus, pinch zoom, etc. This research helps to provide a more accurate environment in augmented reality.

This article demonstrated the design patterns applicability in the augmented reality area. In the current work we present a complex platform with a design to provide a software solution to EV charging platform management.

Omar Bonerge Pineda Lezama et al. [8] build their research on design patterns for the development of an educational application with the purpose of solving the problems that may arise within a team. The group's ability to negotiate reduces the time it takes to find solutions and to implement the designed scheme to analyze and promote teamwork skills.

A. Casteigts et al. [9] formulate multiple algorithms for degree computation, MIS and coloring, using the design patterns such as: peripheral collision detection, adaptive probability, exclusive beeps and multi-slot phases. They work to improve the complexities of the algorithms and prove their convertibility.

Kevin Lano et al. [10] present the transformation applications for the design patterns and the diverse categories of transformation. The research presents the advantages of different patterns usages and it catalogues the new patterns.

In this article, we discuss the advantages and potential drawbacks in software systems development. In order to deliver a clear picture of design patterns topic, we mentioned the difference between the design principle and design pattern, terms that can be easily confused.

We completed the work with a DP classification up to the last updates from technical documentation.

Daniela Fogli, Claudio Greppi and Giovanni Guida [11] analyze the importance of DSS—decision support system—for emergency management, in an extensive variety of application domains. The proposed solution is built on a set of design patterns, the validity of which has been assessed by setting in place an evaluation exercise carried out by expert designers. Dae-Kyoo Kim, Lunjin Lu and Byunghun Lee [12] explore design patterns in terms of consistency and conformance. The authors attempt to demonstrate their perspective on transforming an application model by means of structural and behavioral features of a design pattern, by applying the observer pattern to a graph application.

Figure 1 illustrates some of the issues to be considered while working with the design pattern, as they have been identified in the consulted literature.

Huseyin Ergin, Eugene Syriani and Jeff Gray [13] explore the current literature and highlight 14 existing model transformation design patterns. MODEL TRANSFORMATION seems to be the answer for several problems in MDE—model-driven engineering. The authors propose to employ the Delta language to ensure a comprehensive template meant to represent model transformation design patterns. Irfan Šljivo et al. [14] analyze the role played by design patterns in critical systems design, focusing on safety or security related systems, which need a guarantee with respect to the system's adequacy to perform in a given environment. The authors propose a methodology developed for ensuring the employment of design patterns in such domains.

B. Bafandeh Mayvana, A. Rasoolzadegan and Z. Ghavidel Yazdi [15] have performed a systematic mapping study related to the design patterns, covering the trends, and the gaps in the field that need to be further addressed. They identified six topics in the design patterns field, among which, the most active ones are pattern mining and pattern development.

In the cited manuscript, the authors have carried out extensive research on design pattern debates in technical publications. They have provided a statistic that indicates that the DP subject is an active topic in the last decades through the software engineering specialist. This outcome is not dependent on geographical affiliation, the exploration is present all over the world. These being mentioned, we propose an article that presents a complex overview of design patterns.

In addition, we investigate the Cloud design patterns to establish appropriate cloud architecture to launch the platform in the marketplace. It is essential to remark that solid software structure results are reflected in a stable cloud representation.

Moreover, one important aspect that is considered in this paper refers to ensuring a scalable and configurable cloud application in terms of multi EV charging providers.



Figure 1. Issues of the design patterns.

Thus, a multitenant cloud application is sketching the best possible tenant isolation solution and reveals low-cost resources.

Apostolos Ampatzoglou, Olia Michou and Ioannis Stamelos [16] present a review of the design patterns based on 141 open-source projects, having recorded more than 4500 pattern instances. The built repository has been subjected to an evaluation process, from both an academy and a practical perspective. Zakaria Moudam and Nouredine Chenfour [17] introduce the data management system meant to enable the selection of the most suitable pattern among the ever-growing plethora of design patterns.

Imen Tounsi et al. [18] propose two transformation categories: M2M—standing for model to model—and M2T—meaning model to text. The former enables the transformation of SOA design patterns and the latter facilitates the transformation to Event-B specifications of the compound pattern model, which will now include a formal description and a graphic representation. J. Arm et al. [19] implement and describe a checking monitor based on a runtime model. The monitor uses as a model the extended Petri net, which is implemented with VHDL (VHSIC-HDL, Very High Speed Integrated Circuit Hardware Description Language). The new added feature is runtime checking and it exploits the proposed architecture concept. The aim of the research is to transform the monitor system so it can be integrated in a control system. Angela Patricia Villareal—Freire et al. [20] address the absence of guidelines for designing therapeutic system interface for children’s attention deficit treatment. The authors provide a methodology for extracting patterns through a reverse engineering mechanism, used on Android applications. They identify three major steps in accomplishing this, namely: the applications selection, the patterns selection and the investigation of the patterns present in the chosen applications.

Jiang Li et al. [21] create a novel design pattern assigned to the industrial robot engineering in order to assure their rapid development. The implementation of the user-customized configuration design pattern is dictated by the role played by the various stakeholders of this industry. After establishing the system dynamics models for this type of design pattern and for the traditional one, they proceed to simulating these models by using the Anylogic software. The results returned by the simulation indicate the new proposed design pattern as being more efficient with reference to inventory, and to order response time. Khalid Aljasser [22] highlight the fact that the language used on the design patterns influences the implementation. In the research are compared three types of patterns: decorator, observer and singleton. They use ParaAJ to make the patterns of observer and singleton reusable, but it does not work with the decorator one. This research is the base for future development in the area.

Jing Bai, Haonan Luo and Feiwei Qin [23] propose an innovative approach regarding design pattern extraction in the CAD field. The novel solution consists of three stages: the first one deals with the formation of a relative integrated function resulting from the extraction of reusable regions having the following features: high cohesion, moderate complexity and low coupling; the second step identifies prospect design patterns by means of grouping reusable regions with the help of a specific clustering algorithm. During the last step, patterns with sufficient information and high frequency are selected. This approach proves its effectiveness through experimental results. Seyed Mohammad Hossein Hasheminejad and Saeed Jalili [24] propose a new two-phased method for the selection of the most suitable design pattern, a technique that has proven its efficiency after being employed in several case studies and in real situations. This method relies upon a text classification approach that seeks to indicate to software developers the proper design pattern for dealing with particular problems. Stefan L. Pauwels et al. [25] describe how they have built and verified interaction design patterns meant to function as a guideline in redesigning an application, which would be extremely useful in complex business environments. The above-mentioned patterns were integrated in a pattern language, as a set of rules for human–computer interaction in order to facilitate the further extension of the application. To reach this point, the authors have proficiently applied a series of analytical methods, controlled experiments and user interviews.

Bahareh Bafandeh Mayvan and Abbas Rasoolzadegan [26] present a new two-phased method for detecting design patterns. The first stage focuses on the structural signature of the pattern in terms of semantics and syntax. Following the implementation of a matching algorithm that leads to the identification of candidate instances, the second phase of the method is initiated, and the final matches are established. The technique has proven to be highly efficient and accurate after having been evaluated on various systems, with relevance to precision and recall.

Figure 2 exemplifies the presented solutions for the problems identified in working with design patterns. These solutions were identified in articles and research in the field.

Jae Hyun Lee et al. [27] examine the plug-in electric vehicles charging behavior for 7979 owners. The paper is analyzing the charging location and level of charging to determine their preferences, travel pattern and the charging behavior. The research provides a database with workplace charging availability, commute behavior, vehicle characteristics and sociodemographic information.

Yongxiu He, Qi Zhang and Yuexia Paang [28] project the pattern design of the Chinese electric vehicles depending on the fuel costs, operating costs, initial investment cost and other costs. This paper design three promotion models, which depend on the risk sharing, risk reduction and risk transfer.

Hongqiang Guo et al. [29] are developing a new driving pattern. They verify if the applicability, robustness and reasonability of the method using the Monte Carlo Simulation. The results are showing an improvement of 34.36% on the fuel economy.

Wanying Wan et al. [30] provide a comparative pattern between the private vehicles and the shared ones. The research is presenting from three dimensions the pattern of BEVs

(battery electric vehicles) usage pattern. They are the decision-making dimension, travel spatial and travel temporal one. The differences between the patterns are easily observed by the indicator system and they can be analyzed from multiple perspectives.

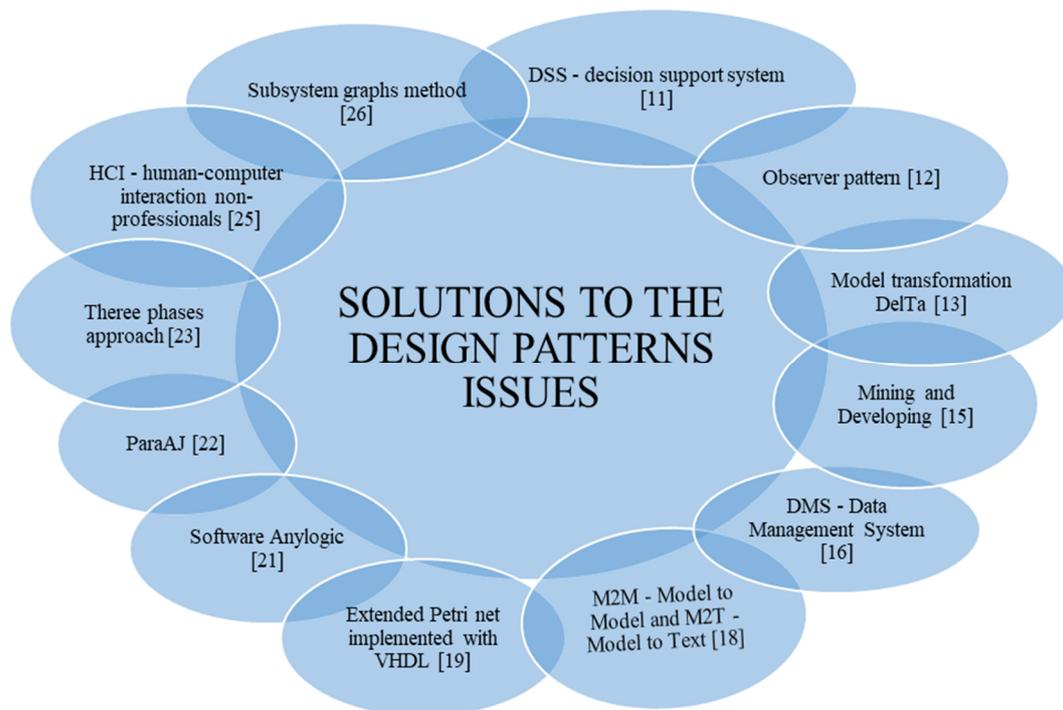


Figure 2. Solutions to the design patterns issues.

Ian Miller, Maryam Arbabzadeh and Emre Gençer [31] present the emissions impact using information from 2018 and 2019. The temperature data, driving and hourly charging creates the charging patterns distributed per region. There are calculated errors and provided emissions approximations in future scenarios.

The previous four manuscripts that are cited in the section above, represent an incursion in the electric vehicle domain. This data will constitute the input in decisions towards the EV platform features and in elaborating the optimal algorithms in charging prediction. Another novel application field for design patterns is related to charging stations, a topical subject for the current evolution of electric vehicles, discussed in detail in [32,33]. Finding the optimal placement for charging stations taking into account the demand and number of electric vehicles was addressed by Raboaca et al. in [34,35]. Similarly, design patterns for a mobile charging station were taken into account by Fodorean et al. in [36].

The AI module is not described in this work but will represent the central point in the future research.

2. Incursion in the Design Patterns Field

Section 2 is an overview of existing software design patterns and their appropriate use cases according to the literature, presenting both the advantages and the potential disadvantages of their use. We also made a clear distinction between design patterns and design principles.

Applying design patterns to software design was first attempted in 1987, by authors Kent Beck and Ward Cunningham. Later, in 1994, four authors released the book *Design Patterns: Elements of Reusable Object-Oriented Software* (aka Gang of Four, aka GoF) [37], which has proven to be essential in the promotion and adoption of the notion in this domain.

To render the above definition less abstract, we proposed to examine the decorator pattern. A decorator “allows behavior to be added to an individual object, either statically

(i.e., at compile time) or dynamically (i.e., at run-time), without affecting the behavior of other objects from the same class" [38]. This is performed by subclassing the original object and at the same time holding a reference to it.

2.1. Design Patterns—Benefits and Potential Drawbacks

We must underline the fact that design patterns are not algorithms. They are arrangements of objects and classes, which smoothly stabilize the clashing forces prone to engender design issues. Software requirements are constantly updated and modified and managing a poorly designed system would represent a real burden for developers. As the system becomes more intricate, the implementation of design patterns proves ever-more useful, becoming an indicator of the system's long-term survival. Moreover, a system made up of objects with tight co-dependencies is harder to navigate and the alteration of any segment is prone to create a snowball effect, generating problems in totally unpredicted areas of the code. Performing tasks such as ensuring modularity and separation of concerns will translate in a considerable decrease of maintenance issues, ensuring, at the same time opportunities for code reuse. Therefore, the success of a software application relies upon its flexibility and resilience to change, so as to meet the constantly evolving expectations and needs, without suffering degradation.

However, according to how and when they are employed, design patterns can represent a real asset in developing software products or they can have significant drawbacks. There are instances when a software designer must trade off the advantages of particular design patterns against the entailed disadvantages, in terms of flexibility and performance.

We could conclude that there are two main benefits of design patterns that cannot be argued. Firstly, they supply proven solutions that have stood the test of time for recurring design problems. This helps to isolate certain system sectors that are susceptible to frequent modifications by promoting architectural principles such as loose coupling or other system properties bearing attributes of resilience, such as modularity. This finally translates into a more understandable, manageable and extendable system.

The second advantage resides in the fact that design patterns ensure a common language, which enables programmers to communicate with ease by referring to a design solution by simply naming a standard pattern, such as decorator, for instance. This makes more sense and is much easier to pinpoint, rather than other lengthy descriptions that might even engender misunderstandings, like: "a wrapper around a base type" or something like "a type that relates to the base type through both a HAS-A and a IS-A relationship". The following statement perfectly illustrates this benefit design patterns bring to the software development community: we will apply decorators for our Streams and then employ builders to smoothly bypass the constructor chaining issue. This is sufficient information for developers to immediately comprehend the necessary steps. Moreover, based on their prior experience with design patterns, developers can easily identify potential failure scenarios.

However, it is a known fact that applying design patterns can increase the complexity of a system and, unfortunately decrease performance, due to their tendency to include additional objects, classes or layers to the design these are applied to. Therefore, the incorrect, unsuitable or at times unnecessary use of design patterns could lead to an excessively complicated code, which would prove difficult to maintain and debug. Moreover, beginners are likely to misunderstand and misuse these extremely abstract construct.

2.2. Distinction between Design Patterns and Design Principles

Before advancing any further, we should make a clear distinction between design patterns and design principles. The latter are more general and high-level than the former, which could be regarded as condensations of principle recipes for the resolution of context-bound design problems. As part of the adaptive programming strategy, a couple of design patterns were proposed for creating a software system that would be easy to extend and maintain over time. These principles are rules created for programmers that need to be

applied while working on software by refactoring the code until it is extensible and eligible. We have provided below some examples of design principles:

- (a) Single-responsibility principle: a class should be appointed only one job to handle, one single reason to change.
- (b) Open-closed principle: object or entities are closed for modifications but opened for extensions.
- (c) Liskov substitution principle: "Let $q(x)$ be a property provable about objects of x of type T . Then $q(y)$ should be provable for objects y of type S where S is a subtype of T ." [39]. This statement implies that all subclasses should be interchangeable with their parent class.
- (d) Interface segregation principle: this principle requires that the software programs should be independent from the interfaces that they do not use.
- (e) KISS (keep it simple stupid): the principle states that the majority of systems work at their best when kept simple, rather than intricate.

Bearing in mind the fact that design principles are high level rules, we proposed to continue with a more in-depth analysis of design patterns. For the purposes of the current paper, we identified three large categories of patterns that could be successfully employed in the development of an EV charging platform: software, cloud and multitenant database design patterns. The following subchapter is dedicated to a thorough analysis of software design patterns, some of which will be further explored in Section 3, along with the other two major types mentioned above, as components of the solution envisaged for the construction of the management platform.

2.3. Software Design Patterns Classification

Design patterns vary in terms of complexity, level of detail and applicability and from these features we can draw up three categories of patterns:

- Creational design patterns deal with the mechanisms of object creation. Their purpose is "to separate a system from how its objects are created, composed and represented. They increase the system's flexibility in terms of the what, who, how and when of object creation" [40].
- Structural patterns deal with identifying simple ways of establishing relationships among objects.
- Behavioral patterns identify and realize common communication patterns between objects.

3. Design Patterns and EV Charging Software

To begin with, Section 3, along with the other two major types mentioned above, as compone provides a brief description of the design patterns categories that will be further discussed in the manuscript.

During our study, we identified three main categories of design patterns used in the development of complex software solutions, namely:

- Software design patterns, which constitute the optimal solutions employed by skilled object-oriented developers to general issues that software developers face during development process.
- Cloud design patterns that have proven valuable for constructing scalable, dependable and secure applications in the cloud.
- Multitenancy model, which assures a stable, isolated, low cost, customizable and scalable cloud architecture for the application's multiple tenants.

These categories are thoroughly explored below in terms of the role they play in the construction of an EV charging management platform.

3.1. Software Design Patterns

Section 3.1 focuses on the main design patterns categories and outlines the base role of DP in a complex software solution development process.

Moreover, the paper proposes a suite of experiments for each DP category. The examples describe the applicability between the communication services, data access layer and in sketching a high-quality coding style. The section ends with an overview of the experiments that have been performed and illustrated.

Design patterns are a general solution to specific scenarios that is commonly met in software design. It can be described as a rule that indicates the interaction between a context, a problem and a solution. This definition has been presented by the software community as a template that provides the necessary information to understand the essence of the structure and the potential flaws of the proposed solution. [37]

A design pattern provides a solution that can be employed in different situations, to address different problems. Each pattern has been designed, implemented and tested in particular circumstances and their reusability has been confirmed; in the development process, when applying DP, we create unit modules that ensure decoupled architecture.

With the advancement of electric vehicles in innovation and industry, and the government's financial motivations and the related policies background, the current electric vehicle effervescence is constantly rising. Increasingly, people choose electric vehicles as a travel option.

This paper focused on building a management platform to facilitate the management process in the EV charging infrastructure.

In the active e-mobility market development, it is imperative to handle the charging process of electric vehicles. The project architecture design has an important role in delivering qualitative software systems. These principles are met in the software communities that produce open-source software application as it is illustrated in Michael Hahsler's article [41]. We cannot envision a development process for an EV charging platform without considering the DP base role. Building project architecture through DP influence, help us to create a software solution based on a solid structure that will easily admit changes or updates in the project flow. It will also allow the unit tests implementation, which is a considerable element in thriving a complex project.

In this solution we considered the microservice implementation and we used DP in building the services instance, in order to assure the services dependencies. Thus, we ensure an efficient allocation of resources and maintain the components decoupled.

In the following sections, we illustrated some design patterns implementations through a brief introduction for each category of software design patterns adopted and we accomplished the presentation with an example from the EV platform project defined through Microservices architecture model. Design patterns vary in terms of complexity, level of detail and applicability and from these features we could draw up three categories of patterns: creational, structural and behavioral. From each category, the design patterns that best answer the needs for creating an EV charging platform have been presented.

3.1.1. Singleton Pattern

This pattern ensures that a single class has only one instance, being at the same time responsible for providing a global point of access to that instance. In the EV charging application development, the microservices communicate using HTTP requests. The service that handles the connection is registered as singleton lifetime and is created the first time they are called upon. Hence, every request will use the same instance.

An example of microservices communication that leans on HTTP calls can be identified between the charging station microservice and the connectors microservice.

The responsibility of the charging station microservice is to provide support in managing the charging points within a network. This microservice will validate the user input in the charging station enrolment and will offer support for managing a charge point network business.

The role of the connectors microservice is to support the registration process and the administration of the EV charging points connectors.

The diagram below further illustrates the communication model between the charging station microservice and the connectors microservice using asynchronous HTTP request/response protocol.

The architectural model for the microservices is based on the Representational state transfer (REST) approach in creating the Web API services. The diagram in Figure 3 presents the charging station microservice communication with the connectors microservices through the Http protocol.

.NET Core framework provides support for the lifetime services registration, once for each client connection. The singleton lifetime is created with the AddSingleton method that requires a specified object type and service implementation (see Figure 4).

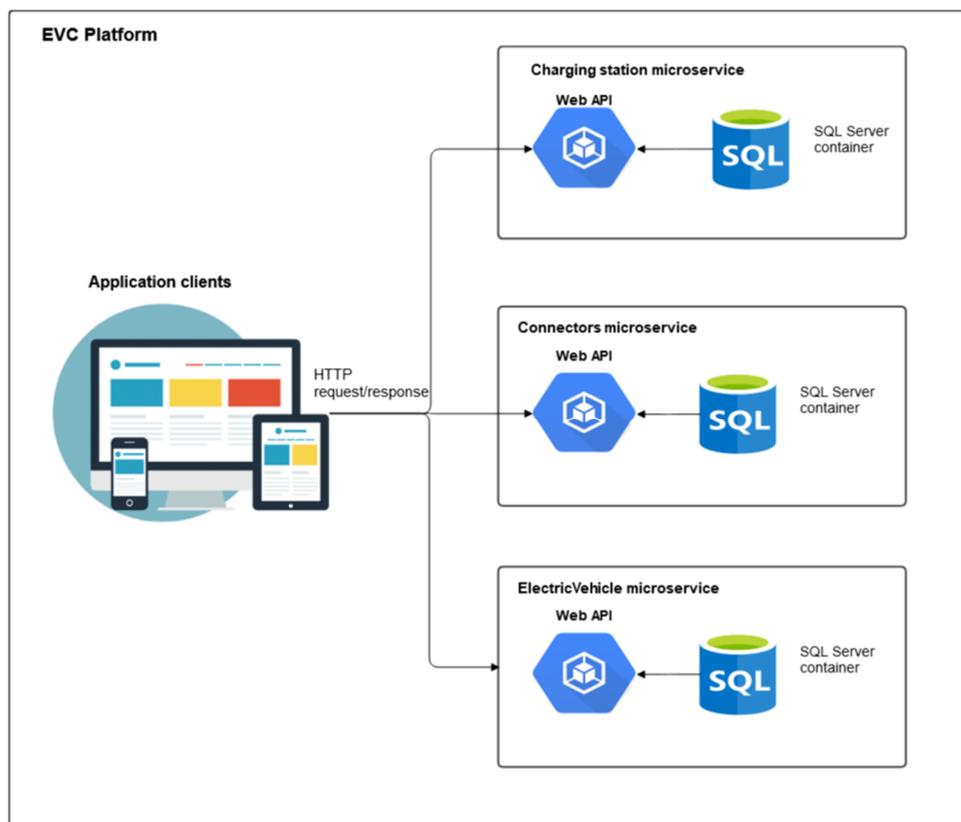


Figure 3. Microservices example using Http request/response communication (synchronous or asynchronous).

```

0 references
public static IServiceCollection AddCoreSingletonCommunicationDependencies(this IServiceCollection services)
{
    var configuration = services.BuildServiceProvider().GetService<IConfiguration>();
    services.Configure<CommunicationConfig>(configuration.GetSection("Communications"));

    services.AddSingleton<IHttpClientService, HttpClientService>();
    services.AddSingleton<IHttpClientRequest, HttpClientRequest>();
    services.AddSingleton<IApiRequest, ApiRequest>();

    return services;
}
    
```

Figure 4. Singleton lifetime services registration.

3.1.2. Dependency Injection Design Pattern

In software engineering, dependency injection (DI) is an object-oriented design pattern whereby one component is independent of its dependencies. It represents the IoC

(Inversion of control) principle implementation and it enables the creation and binding of the dependent objects separately from the services that use them. DI is a programming technique that allows us to implement loosely coupled software components.

In the electric vehicle charging platform, a case where we use DI patterns is the user microservice, which attends to platform users' management.

The user API controller will define and handle the endpoints to capture or update the user profile information. In this pattern implementation we used interfaces and we did not depend on a concrete usage of a service dependency. Instead of instantiating collaborators directly into the constructor, the dependencies are isolated into interfaces and provided as parameters in the constructor.

At the controller level the services that are requested are UserService and logging that contain the business logic and the link to the data access layer.

The capture in Figure 5 represents the dependency injection pattern implementation through a UserProfile controller with user profile and logging services injected in the constructor.

```

namespace SmartEVC.Web.UserApi.Controllers
{
    [AuthorizeAD(AuthRole.All)]
    [ApiController]
    [ApiVersion("1.0")]
    [Route("api/v{version:apiVersion}")]
    [Produces("application/json")]
    3 references
    public sealed class UserProfileController : ControllerBase
    {
        private readonly IUserProfileService userProfileService;
        private readonly ILogService<UserProfileController> log;

        0 references
        public UserProfileController(
            IUserProfileService userProfileService,
            ILogService<UserProfileController> log)
        {
            this.userProfileService = userProfileService;
            this.log = log;
        }

        [HttpGet]
        [Route("user-profiles")]
        1 reference
        public async Task<IActionResult> GetAllAsync()
        {
            try
            {
                var userProfiles = await userProfileService.GetAllAsync();

                return Ok(userProfiles);
            }
            catch (Exception ex)
            {
                await log.ErrorAsync(nameof(GetAllAsync), ex);

                return BadRequest();
            }
        }
    }
}

```

Figure 5. Dependency injection example.

3.1.3. Façade Pattern

The facade structural design pattern delivers a way to decrease the overall complexity of the application by moving the unwanted dependencies to one place. The facade can be easily identified in classes that have simple functionality implemented and delegate a part of the work to other classes. The objective of the facade pattern is to follow the full life cycle of the objects used.

In the next example in Figure 6, the “LogService” class acts as a facade because during the “SaveAsync” method, we built the model to be saved in the cloud and then delegate another microservice to the operation of uploading the information to the external container. Thus, the upload operation can be treated as a separate step, and the implemented functionality can also be used to push binary files into the cloud.

```
namespace SmartEVC.Web.LoggingApi.Services
{
    2 references
    public class LogService : ILogService
    {
        private readonly string blobContainerName;
        private readonly string logFileExtension;
        private readonly IApiRequest apiRequest;
        private readonly string jwt;
        private readonly string currentUserEmail;

        0 references
        public LogService(
            IApiRequest apiRequest,
            IConfiguration configuration,
            IConfigurationService configurationService)
        {
            this.apiRequest = apiRequest;
            blobContainerName = configuration["BlobContainerName"];
            logFileExtension = configuration["LogFileExtension"];
            jwt = configurationService.GetJwt();
            currentUserEmail = configurationService.GetUserEmail() ?? string.Empty;
        }

        2 references
        public async Task SaveAsync(LogEntryDto logEntry)
        {
            var dto = new FileUploadRequest(
                blobContainer: blobContainerName,
                content: Encoding.UTF8.EncodeBase64(logEntry.Message),
                fullName: $"{currentUserEmail}/{logEntry.TimeStamp:dd-MM-yyyy HH:mm:ss}{logFileExtension}");

            await apiRequest.AzureManagementApi.UploadFileAsync(dto, jwt);
        }
    }
}
```

Figure 6. Facade example though LogService implementation.

3.1.4. Repository Pattern

The repository pattern proves useful by creating an “abstraction layer between the data access and the business layer of an application” [42]. Among the advantages of this pattern, we can mention the fact that it ensures code maintainability and reusability. Data access is maintained through specific classes called repositories and the communication with the database is established through a middleware. At its core, it represents a generic repository that will serve us all the CRUD functions, which can be called upon by any of the repository classes in our microservices projects. (CRUD stands for create, read, update and delete)

The scheme in Figure 7 illustrates a repository pattern implementation from the abstract layer to the services usage.

Building the generic repository and repository classes using the respective generic repository does not constitute the final step. We will continue with the creation of a wrapper around the repository classes, called “ChargingStationService”, for instance, and inject it as a service. In this way, the wrapper can be instantiated once and then we call upon any of the repository classes we need within our microservice controllers [42].

The capture in Figure 8 below presents the structure of the abstract layer in the repository pattern practice.

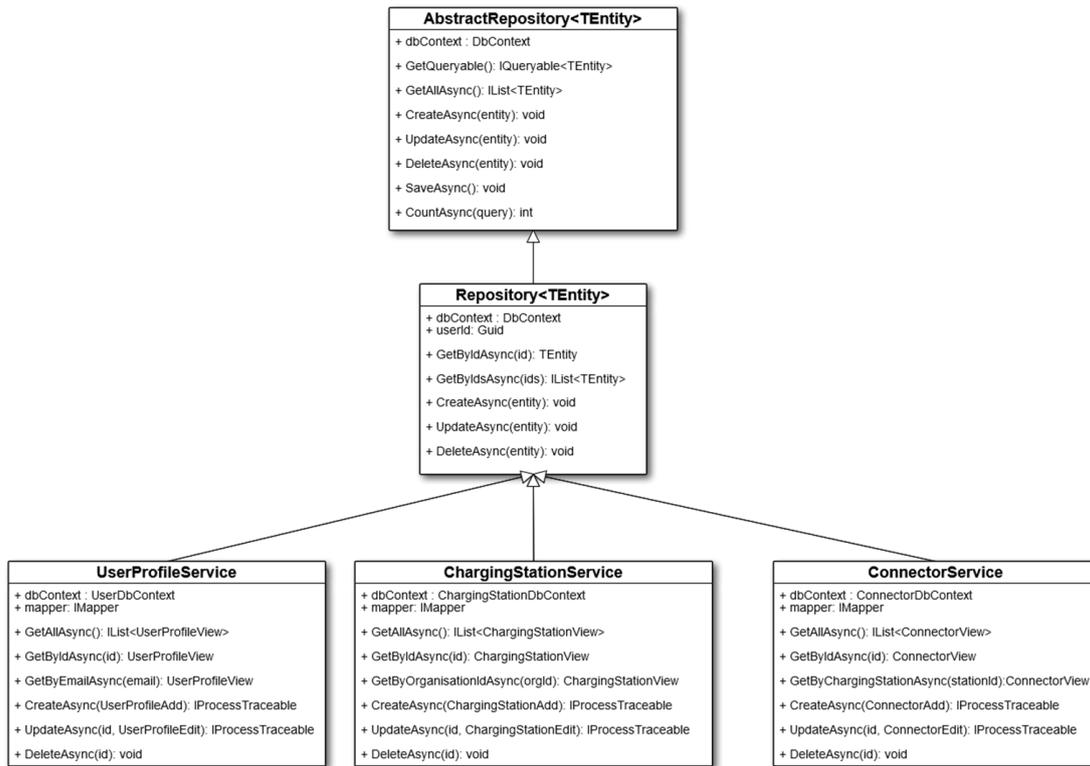


Figure 7. Repository pattern diagram for Electric Vehicle charging platform.

```

namespace SmartEVC.Web.Common.Data.Abstractions.Repository
{
    2 references
    public abstract class AbstractRepository<TEntity> : IAbstractRepository<TEntity>
        where TEntity : class
    {
        private readonly DbContext dbContext;

        1 reference
        public AbstractRepository(DbContext dbContext) {...}

        1 reference
        public virtual IQueryable<TEntity> GetQueryable() {...}

        5 references | 2/2 passing
        public virtual async Task<IList<TEntity>> GetAllAsync(CancellationToken cancellationToken = default) {...}

        5 references
        public virtual async Task CreateAsync(TEntity entity, CancellationToken cancellationToken = default) {...}

        3 references
        public virtual async Task CreateBulkAsync(IList<TEntity> entities, CancellationToken cancellationToken = default) {...}

        6 references
        public virtual async Task UpdateAsync(TEntity entity, CancellationToken cancellationToken = default) {...}

        4 references
        public virtual async Task UpdateBulkAsync(IList<TEntity> entities, CancellationToken cancellationToken = default) {...}

        4 references
        public virtual async Task DeleteAsync(TEntity entity, CancellationToken cancellationToken = default) {...}

        2 references
        public virtual async Task DeleteBulkAsync(IList<TEntity> entities, CancellationToken cancellationToken = default) {...}

        7 references
        public virtual async Task SaveAsync(CancellationToken cancellationToken = default) {...}

        1 reference
        public virtual async Task<int> CountAsync(CancellationToken cancellationToken = default) {...}

        1 reference
        public virtual async Task<int> CountAsync(IQueryable<TEntity> query, CancellationToken cancellationToken = default) {...}
    }
}

```

Figure 8. The implementation of the repository pattern.

3.1.5. Template Method Pattern

The template method is a part of the behavioral design pattern category that defines basic algorithm structure in the superclass, which is usually an abstract one and allows the subclasses that use it to overrule certain steps of the algorithm, keeping its original structure intact.

The idea behind this pattern is to split the algorithm structure into simple steps, which are then turned into methods that are part of the template method. Usually, the steps may have a default implementation or can just be abstract. To be able to use the algorithm, the client needs to provide its own subclass, which implements the abstract methods and overrides some, where needed. So, the template method allows us to break down a monolithic algorithm into a series of specific subclasses and to keep the common functionality into a superclass, eliminating code duplication.

In our case, the template method pattern provides a “skeleton” for various entities of the electrical vehicle platform that needs to be managed by clients via a web portal.

Captured in Figure 9 there is an example of template method pattern applicability through the repository pattern implementation.

```
namespace SmartEVC.Web.Common.Data.Abstractions.Repository
{
    2 references
    public abstract class AbstractRepository<TEntity> : IAbstractRepository<Entity>
        where TEntity : class
    {
        private readonly DbContext dbContext;

        1 reference
        public AbstractRepository(DbContext dbContext) {...}

        1 reference
        public virtual IQueryable<TEntity> GetQueryable() {...}

        5 references | 2/2 passing
        public virtual async Task<IList<TEntity>> GetAllAsync(CancellationToken cancellationToken = default) {...}

        5 references
        public virtual async Task CreateAsync(TEntity entity, CancellationToken cancellationToken = default) {...}

        3 references
        public virtual async Task CreateBulkAsync(ICollection<TEntity> entities, CancellationToken cancellationToken = default) {...}

        6 references
        public virtual async Task UpdateAsync(TEntity entity, CancellationToken cancellationToken = default) {...}

        4 references
        public virtual async Task UpdateBulkAsync(ICollection<TEntity> entities, CancellationToken cancellationToken = default) {...}

        4 references
        public virtual async Task DeleteAsync(TEntity entity, CancellationToken cancellationToken = default) {...}

        2 references
        public virtual async Task DeleteBulkAsync(ICollection<TEntity> entities, CancellationToken cancellationToken = default) {...}

        7 references
        public virtual async Task SaveAsync(CancellationToken cancellationToken = default) {...}

        1 reference
        public virtual async Task<int> CountAsync(CancellationToken cancellationToken = default) {...}

        1 reference
        public virtual async Task<int> CountAsync(IQueryable<TEntity> query, CancellationToken cancellationToken = default) {...}
    }
}
```

Figure 9. Template method implementation in the EV charging platform.

3.1.6. Overview

As we presented in the coding captions in this chapter, design patterns provide methods that can be applied in all the stages of building a software solution.

Thus, among the creational design patterns, we selected the generic repository pattern to define the CRUD functions with respect to the main entities involved, and we completed the services registration using the singleton design pattern. Moreover, in order to ensure that any future requirements will be implemented with ease, we employed the dependency injection technique.

From the category of structural design patterns, we experimented with facade in the development of the logging module, which has the role to ensure an overall view whenever a functionality problem occurs in the application. In line with this practice, we introduced an abstraction layer to isolate the logging methods from its consumers.

For the behavioral component, we explored the template method pattern to define a basic model for other services that confer a high complexity to the algorithm. This way, we ensure the increase of the entire solution complexity throughout a safety approach, in terms of code reusability and organization.

We could confirm that the design patterns concept is placed at the top of best practice within software engineering disciplines.

It brings a major improvement in the development process, especially, in the way that developers are not required to draw up new implementation methods. Design patterns represent templates that cover a technical issue in a specific context and the created design models constitute basic elements in software implementation. Another important aspect is that the design patterns implementation is neutral when transposed to a programming language.

3.2. Cloud Design Patterns

In Section 3.2 we proposed Cloud design patterns as a solution in the software deployment field. After briefly introducing the concept, the study was continued with a presentation of the Cloud DP applicability for hosting the application. The experiments are described through one of the most familiar and structured cloud platforms.

Cloud design patterns differ from the software design patterns and are applied for creating cloud applications, which are defined by reliability, scalability and security. The challenges that must be overcome in cloud development are related to features such as accessibility, data management, messaging, design and implementation, management and monitoring, effectiveness and flexibility, security and resilience.

3.2.1. Publisher/Subscriber Pattern

The publisher/subscriber design pattern impacts the resilience of a system because it allows better communication over the internet in cloud-based and distributed systems, between their components when events occur.

The message must be sent asynchronously, and senders must be decoupled from the consumers. Additionally, a global message queue with a filtering capability must be added. A message is just a data pack, while an event represents a message that informs other constituents about changes or actions that have occurred.

Publisher/subscriber messaging has the following benefits:

- Publishers are loosely coupled to subscribers and can remain oblivious to their existence. Publishers and a subscriber can even be temporarily decoupled.
- Provides scalability and improves responsiveness of the sender.
- Improves reliability.
- It ensures a cleaner integration between systems that employ different platforms, programming languages and communication protocols.

The publish–subscriber model is supported by a series of messaging products and services, among which we recall ServiceBus and EventGrid in Azure.

Figure 10 exemplifies the publish/subscribe diagram [43]. Multiple subscribers receive messages from the publisher using a message broker as a message queue.

Applicability:

For the EV charging station platform, the mobile app user can subscribe, when creating an itinerary, to multiple charging station situated along the way. The user will see in the mobile application or web application if any charging station changed its state and became unavailable with the option to choose another available station close by.

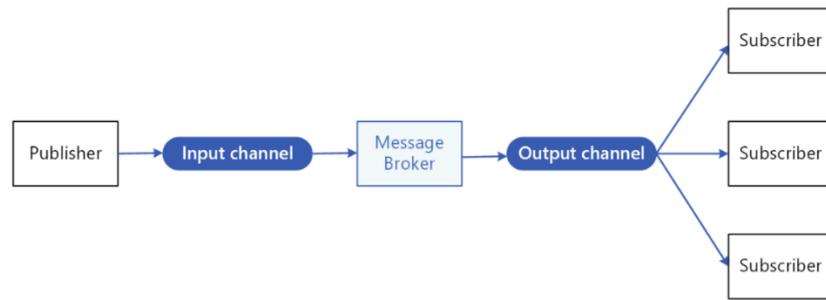


Figure 10. Publisher/subscriber diagram.

3.2.2. Sharding Pattern

Sharding means dividing a data store into a set of horizontal partitions or shards. Shards feature the same schema and hold their own separate subset of the data. A shard is generally comprised of items that fit within a certain range, established by one or more of the characteristics the data possess. These attributes of the data form what is called the shard key or partition key.

Figure 11 Explains a lookup strategy, a sharding logic that implements a map that routes requests for data to the shard that contains the sharding key [44].

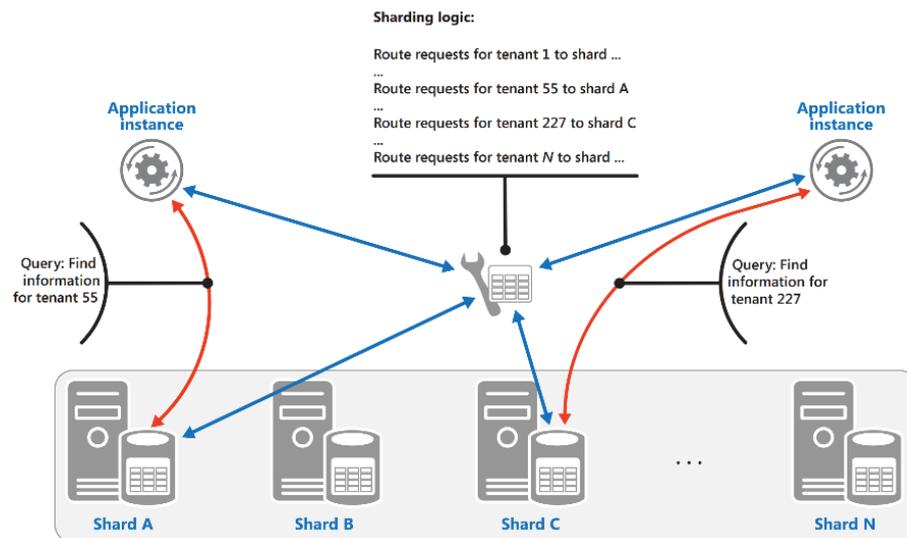


Figure 11. Tenant isolation using shards.

Applicability:

For the EV charging station each tenant will correspond to a shard. Mapping a sharding key to a specific tenant ID must be mapped usually using a complementary database catalog or table.

3.2.3. Static Content Hosting pattern

Using this type of design patterns ensures the deployment of static content to a cloud-based storage service, which can distribute them directly to the customer. In this way, we can experience a decreasing need for expensive compute instances [45].

Applications usually include some static elements like html pages, documents, images that are available to the client and, to be rendered, it uses processing cycles that could often be employed more appropriately. The “storage service can serve requests for these resources, reducing load on the compute resources that handle other web requests. The cost for cloud-hosted storage is typically much less than for compute instances” [46].

This pattern can be employed to:

- Minimize hosting costs for static resources; contingent upon the capabilities of the hosting/cloud provider, entire static websites can be hosted.
- Minimize bandwidth usage. Low bandwidth usage can be achieved using content delivery network, by presenting the cached content from multiple geographical areas datacenter around the world.

Applicability:

EV charging software cloud keep the predefined template for generating reports on the cloud-based storage. The user could generate reports for one or more stations using the web application or mobile phone. The request goes from web app or mobile app to the backend service that will compute the date and use the predefined template from the cloud-based storage to generate the report.

Another way EV charging station could benefit using this pattern is putting the front-end application in a storage account and enabling a content delivery network for the storage account.

This way the front end can get loads of traffic with low cost and without the need of cost or subscription management.

3.2.4. Retry Pattern

The retry pattern allows an application to address passing failures while connecting to a service or to a network resource. This is accomplished by performing a transparent retrial of the operation, presuming that the malfunction is merely transitory. This translates into improved application stability. Therefore, the retry pattern is a resilience category pattern.

An application communicating with elements that run in the cloud need to be fault tolerant for network connectivity issues, temporary service unavailability and timeout that may arise when service is busy.

The application should handle the transient fault elegantly and transparently, minimizing the effects on applications business task.

This pattern should be used only when the application is expected to incur temporary failures, due to that fact it has interactions with remote services or resources. These faults are most likely brief and renewing the request could be successful.

The illustration in Figure 12 on the retry pattern works is explained below:

1. The application calls upon an operation on hosted service and the internal server error is returned (code 500).
2. The application then waits for a short while, before trying again. The failure message is returned once more (code 500).
3. Then, the waiting time increases, before trying once again. The HTTP response is now code 200 (OK) and the request is successful.



Figure 12. Invoking an operation in a hosted service using the retry pattern [46].

Applicability:

EV charging software could benefit from this pattern when it connects the front-end app or the mobile apps to the back end applications.

The retry procedure should be adjusted to meet the business requirements and the nature of the failure:

- Non-critical operations are preferred to fail faster rather than repeating the request several times and affecting the input of the application.
- Aggressive retry policy should have large delays and small number of retries.
- If a request has several failures, maybe it is best to prevent further requests
- Some of the retries may be idempotent (retry logic should handle errors from incomplete requests)
- All retry codes should be fully tested against a wide range of failures conditions.
- All failures should be logged, so that underlying problems can be identified.

3.3. Multitenant Database Design Patterns

A tenancy model establishes the way each tenant data is mapped to storage. It impacts the application design and management. The best suited design pattern for EV charging station system cloud deployment would be a SaaS multitenant application with tenant isolation at database shard or even row level.

There are several design patterns for the cloud, among which we will mention only three that are of particular interest to our study:

- Multitenant app with multitenant databases;
- Multitenant app with database-per-tenant;
- Multitenant app with sharded multitenant databases.

Software as a service (SaaS) is a software distribution model, where a third-party provider hosts applications, rendering them available over the internet. The appropriate tenancy model must be selected to comply with the needs related to:

- Tenant isolation;
- Scalability;
- Cost per tenant;
- Development complexity;
- Operation complexity;
- Customizability.

3.3.1. Multi-Tenant App with Sharded Multitenant Databases

This pattern enables tenant data to be distributed across multiple shards, each shard containing all the data corresponding to one tenant alone.

A sharded model admits almost an unlimited scale.

Figure 13 explains how different tenants are placed in the same database and the isolation between them is done using shards [47]. The catalog database stores the mapping between tenant IDs and the corresponding shards.

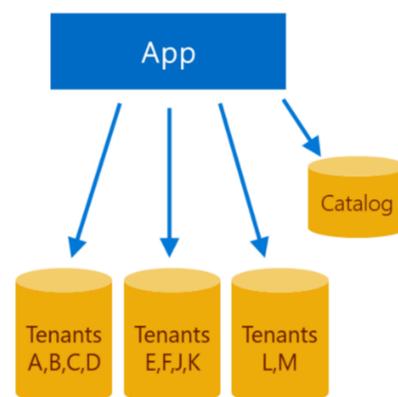


Figure 13. Cloud architecture multitenant schema.

3.3.2. Shards Management

Sharding means increased complexity, in terms of both design and operational management. The mapping between tenants and databases needs to be registered in a catalog. Furthermore, the tenants and the shards have to be managed through the implementation of management procedures. For instance, procedures must be established for adding and removing shards, and for transferring the tenant data between shards.

3.3.3. Shards Management Tools

Elastic database tools for .Net is capable of managing all the tenant and database operations:

1. Managing shards and shard maps;
2. Data-dependent routing;
3. Querying over multiple shards;
4. Adding empty shards;
5. Splitting existing shards;
6. Merging existing shards.

3.3.4. Scaling

Scaling is done through the addition of new shards and their populating with tenants or by splitting a crowded shard into two uncondensed shards.

As described in Figure 14, the shard map manager can be used to easily scale out the database on the Azure SQL Database. The shard map manager is a special type of database having the purpose to store global mapping information in a shard set. The connection of an application to the correct database, contingent upon the sharding key, is ensured by the metadata. In addition, maps tracking local shard data (shardlets) are enclosed in every shard in the set.

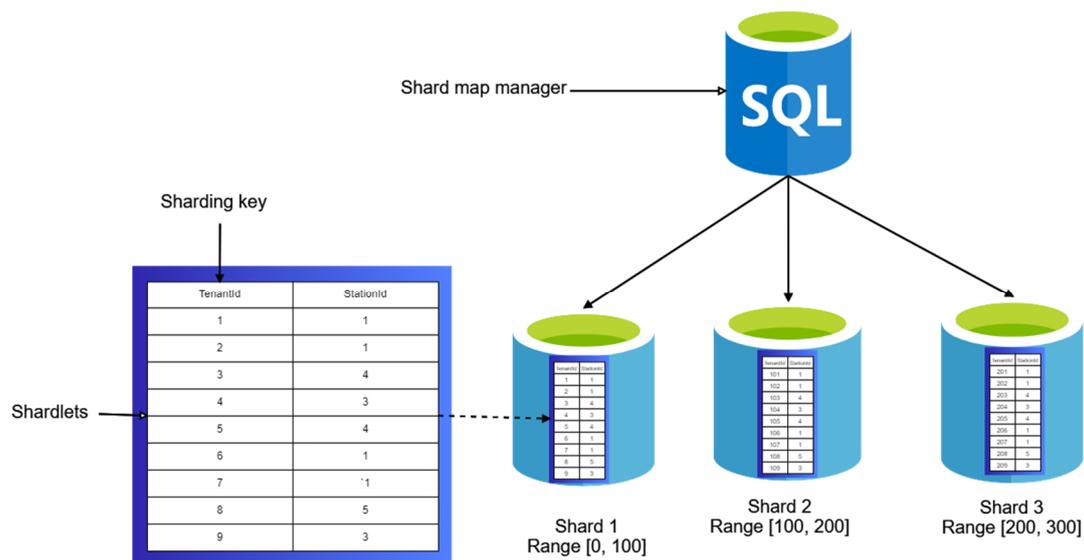


Figure 14. Global mapping information using the shard map manager.

3.3.5. Data-Dependent Routing

The purpose is to employ the elastic database client library data-dependent routing APIs to ensure the automatic connection of each tenant to their assigned shard database. One shard alone holds the TenantId value appointed to a particular tenant, the TenantID representing the sharding key. Once the connection is achieved, an RLS security policy guarantees that any given tenant cannot gain access to any other data rows, other than the ones containing their TenantID.

3.3.6. Row Level Security

RLS (row level security) allows for the safe storage of data for multiple tenants in the same database table. Using security policies, it can add tenant isolation because it filters out rows belonging to other tenants using a database query. Row level security may be useful when there is only a small amount of data related to a tenant and sharding might not be a viable option. Row level security is a viable option instead of using code to enforce security.

Figure 15 illustrates the data dependent routing APIs connection to the database [48]. In the same shards we could place different tenant data that can be isolated using row level security.

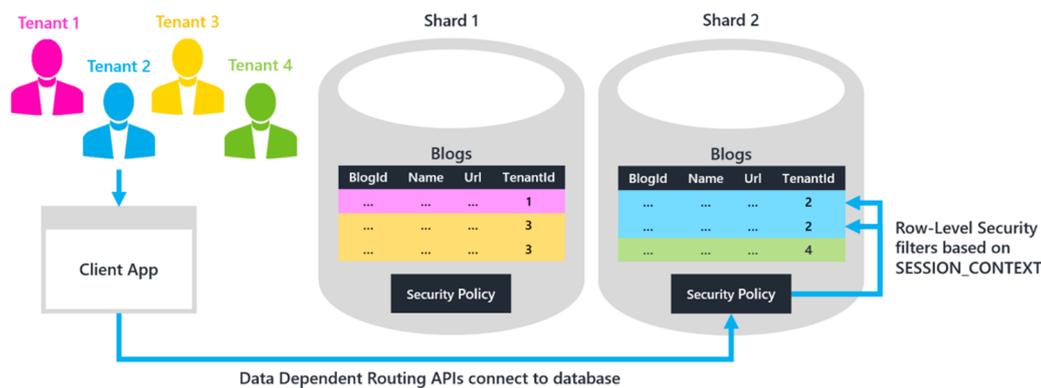


Figure 15. Cloud multitenant architecture isolation at the shard and row level.

4. Results and Discussion

Section 4 contains the results of this work and highlights the benefits of using design patterns as a key in the software development strategy.

These methods still represent a major place in the software industry by influencing project development from team communication to important aspects of software delivery.

Furthermore, this section makes references to the future work regarding the open charge point protocol.

Design patterns, in general, could speed up the software product time to market with proven solutions to a well-known architectural or software problem by shortening the implementation, integration and approval time and facilitating communication between development teams.

Using the multitenant database design pattern, we can achieve a scalable, secure, multitenant, cost effective cloud application with medium development complexity due to sharding. Integrating it with continuous integration/continuous delivery DevOps pipelines will speed up the delivery and deployment of the software product providing a fully automated workflow.

Using the cloud design patterns we can architect a great distributed application that implies security, resiliency, performance and scalability and availability.

Software design patterns exemplified in the previous chapters provide a clearer picture of the implemented design, low coupling, a better coding standard, easier to be tested and code reusability.

The articles that we studied debate a specific category of design patterns and the applicability has been proven in different domains, such as educational applications [8] or emergency management systems [11]. The current manuscript describes the design patterns in every step of platform development and mentions the role for each category of software, cloud and tenancy model design patterns. Thus, it illustrates the harmony and the collaboration of the design patterns category as an important direction in software solution development. More precisely, in building the software architecture, software design patterns have represented the foundation for optimal solution elaboration. We

completed the study in this field, and we provided in Section 3, examples from the project that we were currently working on. The experiments were implemented according to the articles that were mentioned in the “State of the art” section and were validated with the technical documentation from Microsoft, which is constantly updated.

In the case of EV charging software, the cloud design pattern helped to achieve a fast and automatic tenant management. Every new client or tenant receives its own database shard and cloud resources already prepared, on registration, along with cost-effective cloud architecture, with almost limitless scaling providing good isolation between tenants.

Cloud design patterns have represented the main point in creating Cloud architecture for the platform publishing plan. In Section 3 we elaborated those design patterns with direct reference to the EV charging stations platform. The handling of Cloud design patterns applicability in the Cloud structure of this application was strongly emphasized with a description, diagrams and with an applicability segment.

The multitenancy model found its applicability harmoniously in the EV platform when we considered multiple charging stations providers. In this case, each provider will have its own resources in the cloud. Thus, this cloud pattern assured a stable, isolated, low cost, customizable and scalable cloud architecture for the application’s multiple tenants.

Software design patterns fill the need for a robust code base, enforcing a better coding standard and seamless integration with the OCPP, JSON over web sockets technology, which represents the core part of the EV charging software.

For the EV charging cloud application the architecture is really important because the requests start from the front end or mobile apps to the back end microservices and from here it will access the charging station. For this application monitoring, resilience, fault tolerance, scalability and availability in the production environment is mandatory.

5. Future Research Directions

Section 5 presents future research on the electric vehicle charging management platform.

In future work, we will consider design patterns for enabling a high level of maintainability of blockchain based applications such as utility or security tokens, wallets and interfaces for decentralized networks.

We will maintain the use of design patterns as a model in the EV charging platform and other complex projects, as an important pencil in the art of software development. They are a well-documented field in the software industry and are flexible and compatible with any type of software architecture style.

Moreover, we will continue the discussion about the electric vehicle charging management with impact towards the communication protocol and designing a prediction model for EV charging during a journey.

6. Conclusions

Section 6 contains the conclusions and policy implications.

In this work, we discussed and summarized the design patterns applicability in the Smart EVC project architectural model.

Smart EVC solution is a low-cost SaaS (software as a service) solution, providing an easy way to enroll as a business or as a tenant and start using the platform service with your own charging stations and clients or users. Single database for multiple clients or tenants, together with sharing the same software, using containers for deployment provides reliability, scalability and availability at the lowest cost possible and we found this as a limitation to the previous works. The entire system is capable of autoscaling, updating without down-time and being reliable and resilient on a heavy workload.

We started with a formal introduction in the DP area and a review of earlier studies on the use of design patterns in software development. We went on to provide a brief classification, highlighting the pros and cons of each design pattern taken into consideration for the purpose of this paper.

Despite the disadvantages of design patterns presented in previous chapters, the benefits their use provides in the development process, in terms of team communication and final product quality overcome limitations.

We then proposed to continue the research of design patterns implementation in software design by carrying out a case study related to the electric vehicle charging software platform.

The integration of a series of design patterns, namely singleton, factory, repository, observer and factory method template patterns yielded promising results in constructing a solid base structure for the API. The system's functionality will be further evaluated and validated in the course of time.

The examples are written in the .NET Core 3.1 framework and this describes the services communication over the EV charging microservices system.

Towards the end of the research, we explored the multitenant design patterns in the deployment process to ensure a proper tenancy model for a multitenant application.

Author Contributions: Conceptualization, C.F. and A.O. Methodology, A.O., M.M.; Software, M.M., A.M., Validation, A.O., M.M. and A.T.; Formal Analysis, M.M., A.O., A.M., A.V.; Investigation, M.M., A.O., A.M., A.V.; Resources, M.M., A.M., A.T., A.V.; Data Curation, M.M., A.T., A.V.; Writing—Original Draft Preparation, M.M., M.S.R. and A.M.; Writing—Review and Editing, M.M. and M.S.R.; Visualization, A.O., M.M., A.V., A.T.; Supervision, A.O., M.M.; Project Administration, A.O. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by a grant of the Romanian Ministry of Education and Research, CCCDI—UEFISCDI, project number 40PTE of 01.06.2020, within PNCDI III.

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

References

1. Amato, F.; Moscato, F. Model transformations of MapReduce Design Patterns for automatic development and verification. *J. Parallel Distrib. Comput.* **2017**, *110*, 52–59. [[CrossRef](#)]
2. Cacho, N.; Sant'anna, C.; Figueiredo, E.; Dantas, F.; Garcia, A.; Batista, T. Blending design patterns with aspects: A quantitative study. *J. Syst. Softw.* **2014**, *98*, 117–139. [[CrossRef](#)]
3. Fojtik, R. Design patterns in the teaching of programming. *J. Procedia Soc. Behav. Sci.* **2014**, *143*, 352–357. [[CrossRef](#)]
4. Van Diggelen, W.; Overdijk, M. Grounded design: Design patterns as the link between theory and practice. *Comput. Hum. Behav.* **2009**, *25*, 1056–1066. [[CrossRef](#)]
5. Hussain, S.; Keung, J.; Khan, A.A. Software design patterns classification and selection using text categorization approach. *Appl. Soft Comput.* **2017**, *58*, 225–244. [[CrossRef](#)]
6. Arcelli, D.; Di Pompeo, D. Applying Design Patterns to Remove Software Performance Antipatterns: A Preliminary Approach. *Procedia Comput. Sci.* **2017**, *109*, 521–528. [[CrossRef](#)]
7. ElSayed, N.A.; Smith, R.T.; Marriott, K.; Thomas, B.H. Context-aware design pattern for situated analytics: Blended Model View Controller. *J. Vis. Lang. Comput.* **2018**, *44*, 1–12. [[CrossRef](#)]
8. Lezama, O.B.; Manotas, E.N.; Mercado-Caruzo, N. Analysis of design patterns for educational application development: Serious Games. *Procedia Comput. Sci.* **2020**, *175*, 641–646. [[CrossRef](#)]
9. Casteigts, A.; Métivier, Y.; Robson, J.M.; Zemmari, A. Design patterns in beeping algorithms: Examples, emulation, and analysis. *Inf. Comput.* **2019**, *264*, 32–51. [[CrossRef](#)]
10. Lano, K.; Kolahdouz-Rahimi, S.; Yassipour-Tehrani, S.; Sharbaf, M. A survey of model transformation design patterns in practice. *J. Syst. Softw.* **2018**, *140*, 48–73. [[CrossRef](#)]
11. Fogli, D.; Greppi, C.; Guida, G. Design patterns for emergency management: An exercise in reflective practice. *Inf. Manag.* **2017**, *54*, 971–986. [[CrossRef](#)]
12. Kim, D.-K.; Lu, L.; Lee, B. Design pattern-based model transformation supported by QVT. *J. Syst. Softw.* **2017**, *125*, 289–308. [[CrossRef](#)]
13. Ergin, H.; Syriani, E.; Gray, J. Design pattern oriented development of model transformations. *Comput. Lang. Syst. Struct.* **2016**, *46*, 106–139. [[CrossRef](#)]
14. Šljivo, I.; Uriagereka, G.J.; Puri, S.; Gallina, B. Guiding assurance of architectural design patterns for critical applications. *J. Syst. Archit.* **2020**, *110*, 101765. [[CrossRef](#)]
15. Mayvan, B.B.; Rasoolzadegan, A.; Yazdi, Z.G. The state of the art on design patterns: A systematic mapping of the literature. *J. Syst. Softw.* **2017**, *125*, 93–118. [[CrossRef](#)]
16. Ampatzoglou, A.; Michou, O.; Stamelos, I. Building and mining a repository of design pattern instances: Practical and research benefits. *Entertain. Comput.* **2013**, *4*, 131–142. [[CrossRef](#)]

17. Moudam, Z.; Chenfour, N. Design Pattern Support System: Help Making Decision in the Choice of Appropriate Pattern. *Procedia Technol.* **2012**, *4*, 355–359. [[CrossRef](#)]
18. Tounsi, I.; Kacem, M.H.; Kacem, A.H.; Drira, K. Transformation of compound SOA Design Patterns. *Procedia Comput. Sci.* **2017**, *109*, 408–415. [[CrossRef](#)]
19. Arm, J.; Bradac, Z.; Bastan, O.; Streit, J.; Misik, S. Design pattern for the runtime model-based checking of a real-time embedded system. *Ifac Pap.* **2019**, *52*, 127–132. [[CrossRef](#)]
20. Villareal-Freire, A.P.; Aguirre, A.F.A.; Ordoñez, C.A.C. Reverse engineering for the design patterns extraction of android mobile applications for attention deficit disorder. *Comput. Stand. Interfaces* **2019**, *61*, 147–153. [[CrossRef](#)]
21. Li, J.; Daaboul, J.; Tong, S.; Bosch-Mauchand, M.; Eynard, B. A design pattern for industrial robot: User-customized configuration engineering. *Robot. Comput. Integr. Manuf.* **2015**, *31*, 30–39. [[CrossRef](#)]
22. Aljasser, K. Implementing design patterns as parametric aspects using ParaAJ: The case of the singleton, observer, and decorator design patterns. *Comput. Lang. Syst. Struct.* **2016**, *45*, 1–15. [[CrossRef](#)]
23. Bai, J.; Luo, H.; Qin, F. Design pattern modeling and extraction for CAD models. *Adv. Eng. Softw.* **2016**, *93*, 30–43. [[CrossRef](#)]
24. Hasheminejad, S.M.H.; Jalili, S. Design patterns selection: An automatic two-phase method. *J. Syst. Softw.* **2012**, *85*, 408–424. [[CrossRef](#)]
25. Pauwels, S.L.; Hübscher, C.; Bargas-Avila, J.A.; Opwis, K. Building an interaction design pattern language: A case study. *Comput. Hum. Behav.* **2010**, *26*, 452–463. [[CrossRef](#)]
26. Mayvan, B.B.; Rasoolzadegan, A. Design pattern detection based on the graph theory. *Knowl. Based Syst.* **2017**, *120*, 211–225. [[CrossRef](#)]
27. Lee, J.H.; Chakraborty, D.; Hardman, S.J.; Tal, G. Exploring electric vehicle charging patterns: Mixed usage of charging infrastructure. *Transp. Res. Part D Transp. Environ.* **2020**, *79*, 102249. [[CrossRef](#)]
28. He, Y.; Zhang, Q.; Pang, Y. The development pattern design of Chinese electric vehicles based on the analysis of the critical price of the life cycle cost. *Energy Policy* **2017**, *109*, 382–388. [[CrossRef](#)]
29. Guo, H.; Hou, D.; Du, S.; Zhao, L.; Wu, J.; Yan, N. A driving pattern recognition-based energy management for plug-in hybrid electric bus to counter the noise of stochastic vehicle mass. *Energy* **2020**, *198*, 117289. [[CrossRef](#)]
30. Miller, I.; Arbabzadeh, M.; Gencer, E. Hourly Power grid variation, electric vehicle charging patterns and operating emission, ACS Publication. *Environ. Sci. Technol.* **2020**. [[CrossRef](#)]
31. Hilton, G.; Bryden, T.; de Leon, C.P.; Cruden, A. Dynamic charging algorithm for energy storage devices at high rate EV chargers for integration of solar energy. *Energy Procedia* **2018**, *151*, 2–6. [[CrossRef](#)]
32. Rata, M.; Rata, G.; Filote, C.; Raboaca, M.S.; Graur, A.; Afanasov, C.; Felseghi, A.R. The Electrical Vehicle Simulator for Charging Station in Mode 3 of IEC 61851-1 Standard. *Energies* **2020**, *13*, 176. [[CrossRef](#)]
33. Badea, G.; Felseghi, R.-A.; Varlam, M.; Filote, C.; Culcer, M.; Iliescu, M.; Răboacă, M.S. Design and Simulation of Romanian Solar Energy Charging Station for Electric Vehicles. *Energies* **2019**, *12*, 74. [[CrossRef](#)]
34. Raboaca, M.S.; Filote, C.; Bancescu, I.; Iliescu, M.; Culcer, M.; Carlea, F.; Lavric, A.; Manta, I.; Lungu, F.; Bisoc, A.L. Simulation of a Mobile Charging Station Operational Mode Based on Ramnicu Valcea Area. *J. Prog. Cryog. Isot. Sep.* **2019**, *22*, 45–54.
35. Fodorean, D.; Cirlea, F.; Raboaca, M.S.; Filote, C. New mobile charging station for urban and resort areas. In Proceedings of the 2019 Electric Vehicles International Conference (EV), Bucharest, Romania, 3–4 October 2019; pp. 1–6.
36. Răboacă, M.S.; Băncescu, I.; Preda, V.; Bizon, N. Optimization Model for the Temporary Locations of Mobile Charging Stations. *Mathematics* **2020**, *8*, 453. [[CrossRef](#)]
37. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.; Patterns, D. *Design Patterns—Elements of Reusable Object-Oriented Software*; Addison-Wesley: Boston, MA, USA, 1994.
38. Wikipedia. Available online: https://en.wikipedia.org/wiki/Decorator_pattern (accessed on 21 October 2020).
39. Wikipedia. Available online: https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#dependency-inversion-principle (accessed on 22 October 2020).
40. Wikipedia. Available online: https://en.wikipedia.org/wiki/Creational_pattern (accessed on 21 October 2020).
41. Available online: https://michael.hahsler.net/research/patterns_oss2004/OSS_patterns_preprint.pdf (accessed on 27 October 2020).
42. Available online: <https://code-maze.com/net-core-web-development-part4/> (accessed on 3 November 2020).
43. Available online: <https://docs.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber> (accessed on 3 November 2020).
44. Available online: <https://docs.microsoft.com/en-us/azure/architecture/patterns/sharding> (accessed on 3 November 2020).
45. Available online: <https://docs.microsoft.com/en-us/azure/architecture/patterns/static-content-hosting> (accessed on 3 November 2020).
46. Available online: <https://docs.microsoft.com/en-us/azure/architecture/patterns/retry> (accessed on 4 November 2020).
47. Available online: <https://docs.microsoft.com/en-us/azure/azure-sql/database/saas-tenancy-app-design-patterns> (accessed on 4 November 2020).
48. Available online: <https://docs.microsoft.com/en-us/azure/azure-sql/database/saas-tenancy-elastic-tools-multi-tenant-row-level-security> (accessed on 4 November 2020).