Academic year
2022 - 2023

Software Design: using Normalised Systems Theory versus Refactoring

Beau De Clercq

Master's thesis
Master of Science in computer science: software engineering

Supervisors
prof. S. Demeyer, UAntwerpen
prof. H. Mannaert, UAntwerpen

University of Antwerp
Faculty of Science

Contents

Abstract

Software Design: using Normalised Systems Theory versus Refactoring

Beau De Clercq

1. Introduction

In the current software landscape, properties such as maintainability, extensibility and stability are highly valued and necessary to avoid code rot[Sof23]. In this section we will briefly go over what each of these properties mean, why it is important in a software system and how we can find and solve related problems.

1.1. Stability

Software stability refers to the reliability and robustness of a system, which ensures that it performs consistently and predictably under various conditions without crashing or producing unexpected errors. It is a critical aspect of software quality, as unstable software can lead to data loss, system downtime, and user dissatisfaction.
One key factor in achieving software stability is rigorous testing[sta21]. Robust testing processes, including unit testing, integration testing, and regression testing, help identify and fix bugs and vulnerabilities before the software is released to users. Testing also helps uncover edge cases, unusual inputs, and stress conditions that could cause instability.
Another important aspect of software stability is design and architecture. Well-designed software with a clear architecture and modular components is less likely to develop stability issues. Good design practices, such as separation of concerns, loose coupling, and high cohesion, contribute to the stability of software systems. Additionally, adhering to coding best practices, such as error handling, exception handling, and defensive programming, can help prevent software instability caused by unexpected situations.

1.2. Maintainability

The maintainability of a software system refers to the ease with which it can be maintained, i.e., how easy is it to keep the system running and up to date. This is important since these days the Agile development process is most often used, leading to a near continuous cycle[Pie] of maintaining, updating and upgrading. If software becomes unmaintainable the entire workflow is at risk of getting disrupted.
As the lifecycle of a software system goes on, due to added functionalities and improvements, the overall code complexity increases and with increased complexity comes an increase in the amount of time needed to maintain the system. Another factor that leads to this increase in maintenance time is technical debt.
Technical debt[tec][sof] is a term that refers to the cost you will pay in the future by trying to realise a quick fix in the present. This can happen for instance when programmers are under pressure to deliver a specific piece of software, resulting in corners being cut or general low code quality. When this happens there often is no plan as to when the debt will be re-payed and debt will silently accumulate as time goes on. The best way to counter technical debt is to keep in mind the long term effects certain changes may have.
A very nice analogy for software and maintenance over time are plants. When you first buy a plant it is

compact and well defined, but within a few months this compact plant starts growing in every available direction - the addition of new functionalities and improvements of existing functionalities. And while there are no issues at first, you might notice that in some parts your plant is really getting out of hand - the accumulation of technical debt and general code deterioration. So in an effort to keep your plant maintainable you clip the problematic branches - or in the case of software, you start refactoring.

1.3. Extensibility

Extensibility[Ext21][Bab21] refers to ease with which a system or application can be extended - i.e. updated, customized or upgraded - without needing to make large alterations to the existing codebase. One common way to achieve software extensibility is by working with APIs, i.e. interfaces and protocols that allow external components or plugins to interact with the software.

One of the key benefits of software extensibility is the ability to adapt and evolve software over time. As requirements change or new technologies emerge, software can be extended to accommodate these changes without requiring a complete rewrite or overhaul. This enables software applications to remain relevant and competitive in a fast-paced and ever-changing technology landscape. Additionally, software extensibility promotes collaboration and innovation by allowing third-party developers to contribute new functionalities or integrate with existing software, fostering a rich ecosystem of plugins, extensions, or integrations that can enhance the overall value and utility of the software.

Another advantage of software extensibility is its potential to empower end-users to customize software to their specific needs. With extensible software, users can customize the functionality, appearance, or behavior of the software to align with their unique requirements or preferences. This can result in a more personalized and user-friendly experience, leading to increased user satisfaction. Moreover, software extensibility can enable users to create their own plugins or extensions, fostering a community-driven approach to software development and fostering a sense of ownership and empowerment among users.

2. Research question

The goal of this thesis is to look at the feasibility of refactoring a legacy system such that they correspond to NST and verify that both the maintainability and the extensibility of the system have improved. To this end we will:

- Take a (legacy) system and refactor it using a selected method such that the system complies to the NST principles.
- Add new features to both the old and refactored system and compare the overall ease of implementing these new features as well as the code health determined by SonarQube.

In what follows we make a few assumptions:

- Refactoring a system such that it corresponds to NST should be possible in a reasonable amount of time.
- Once refactored, adding new features to the system should be much easier. (= Extensibility)
- Once refactored, updating existing features should be as easy or easier (= Mainainability).
- Once refactored, there should not be a negative impact on the performance of the system.

3. Defining the parameters

3.1. Software Architecture

Just as with the physical type, a software architecture is the entirety of all components and their placements needed to build a product. To do so in the most efficient way possible, over the years numerous architectural patterns have been developed to provide clear guidelines on how to solve a certain set of problems. In addition to these architectural patterns, a multitude of design patterns have been developed to provide a clear structure to each of the components of the architectural pattern.

3.1.1. Architectural patterns

Architectural patterns define the foundation on which the software will be build: they provide general solutions to recurring problems. Among the more well known are client-server, multi-layer, microservices and REST.
It is perfectly possible to use architectural patterns in combination with one another. For example, in the microservice pattern a developer could decide to implement one service according to the multi-layer pattern, e.g. an application to rent a car, another one with REST, e.g. the front-end to the rental application, and yet another one as client-server, e.g. the location were all invoices are stored.

3.1.1.1. Client-server

In the client-server pattern, as the name suggests, we have clients which communicate with servers such as a web server or file server. Whether a computer is a client, a server, or both, is determined by the nature of the application that requires the service functions. For example, a single computer can run a web server and file server software at the same time to serve different data to clients making different kinds of requests. The client software can also communicate with server software within the same computer.
In order to support frequent/high traffic, the application is often run on multiple servers. Clients then connect to one of the servers through a load balancer. This load balancer distributes all incoming client traffic among the available servers, thus ensuring that no server receives more than it can handle. An example of how a load balancer can be placed is shown in Figure 3.1[NGI].

3.1.1.2. Multi-layer

A multi-layer or multitier architecture is a client-server architecture in which functionalities are physically separated over multiple layers. The most common layer division[Ric15] is Presentation-Business-Persistance-Database. An example is shown in Figure 3.2[Ric15]

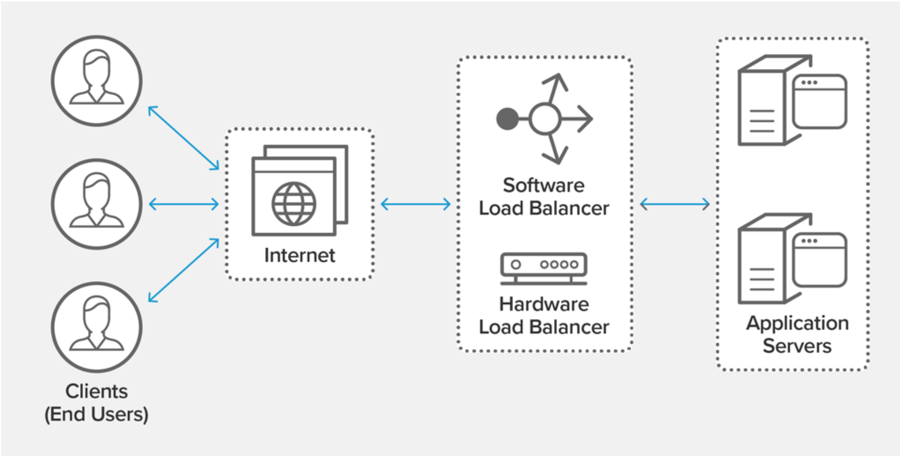

Figure 3.1.: Load balancing diagram

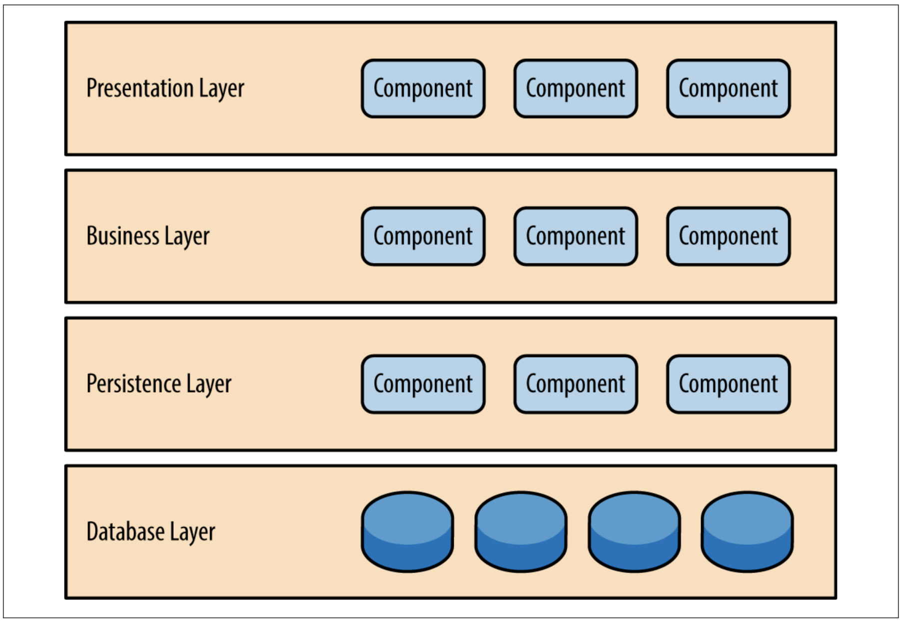

Figure 3.2.: Example of multi-tier architecture

One of the powerful features of the layered architecture pattern is separation of concerns among components. Components within a specific layer deal only with logic that belongs to that layer. For example, components in the presentation layer deal only with presentation logic, whereas components in the business layer deal only with business logic.
In this architecture each layer is also closed, meaning that as a request moves from layer to layer, it must go through the layer right below it to get to the next layer below that one. The reason we don't allow each layer to have direct access to all others is due to the concept layers of isolation. This concept means that if changes are made in one of the layers, these changes don't impact or affect the other layers. This concept also means that each layer is independent of the rest, thereby having no knowledge of how the other layers work internally.

3.1.1.3. Microservices

The microservice architectural style is an approach to developing a single application as a suite of small services where each service runs in its own process and can communicate via lightweight mechanisms. Microservices are built around business capabilities and are independently deployable by fully automated deployment machinery. They can be written in different programming languages and can use different data storage technologies. When using microservices there is only the bare minimum of centralized management.
As an example, consider a webshop as depicted in Figure 3.3. In this example our application consists

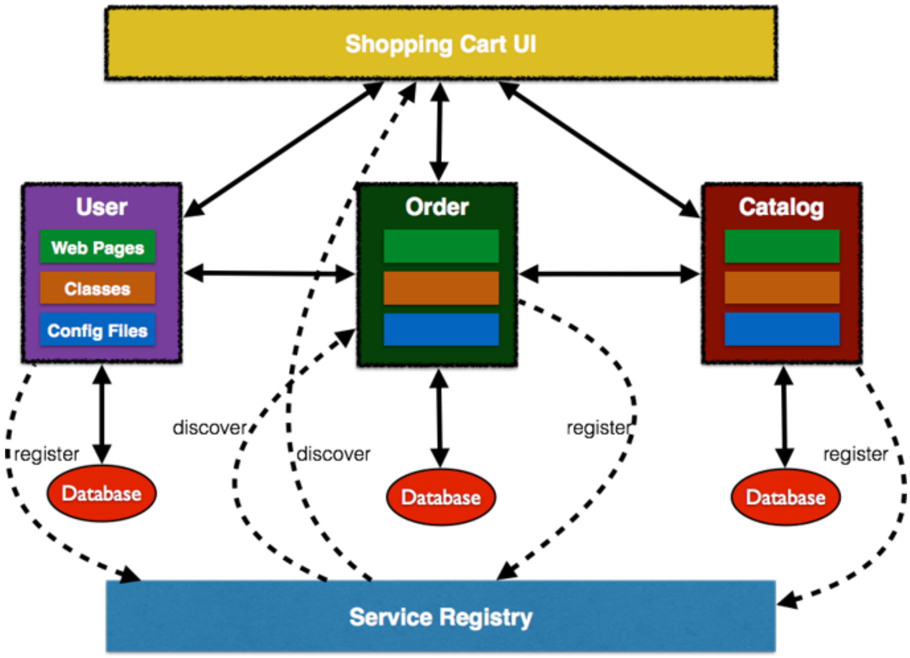

Figure 3.3.: Example of microservice architecture

of 3 components: Catalog, Order, User. Each component has it's own databases so that each microservice can evolve and choose whatever type of datastore – relational, NoSQL, flat file, in-memory or some thing else – is most appropriate. Each component will register with a Service Registry. This is required because multiple stateless instances of each service might be running at a given time and their exact endpoint location will be known only at the runtime. Client interaction for the application is defined in another application, Shopping Cart UI in our case. This application mostly discover the services from Service Registry and composes them together. It should mostly be a dumb proxy where the UI pages of different components are invoked to show the interface.
At time of writing, the most well-known and most used framework for microservices is Docker. Docker works by running software in containers which can then communicate with each other via API calls. Each container can then be developed separately.

3.1.1.4. REST

Representational state transfer[ECPB12] (REST) is a style of software **architecture** for distributed systems such as the World Wide Web.
The term resource is a central concept that denotes any item of interest identified by some identifier. An identifier, also called Uniform Resource Identifier (URI), can be either a URL[1] (the location where you

[1]Uniform Resource Locator

can find the object or the method for finding it) or a URN[2] (which defines an item identity).
A RESTful API is a Web Service that adheres to the REST style, meaning

- It has an Internet media type for data (XML, JSON, ...)
- It uses standard HTTP methods (GET, PUT, POST, DELETE)
- It has a base URI
- It uses hypertext links to reference state
- It uses hypertext links to reference related resources

The REST style states six constraints which, if adhered to, provide an architecture which has properties such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability. These constraints are as follows:

- Client–server architecture: employing the client-server architecture enforces the principle of separation of concerns by separating UI from data storage, allowing components to evolve independently.
- Statelessness: when communicating with a server, each packet send by the client can be understood separately from the others as it will contain all relevant session data, i.e. no session data is kept by the server.
- Cacheability: responses received by the clienst must identify thenselves as either cacheable or non-cacheable, this to prevent clients from providing stale or inappropriate data in response to further requests. If used correctly, caching can in part or fully eliminate some client-server interactions.
- Layered system: clients should not be able to tell whether it is connected to an end-server directly or to a proxy, i.e. the use of load balancers and the addition of security layers won't impact client communications.
- Uniform interface: a uniform interface enables the architecture to be decoupled. It's constraints are the following:
  - Resource identification in requests: resources are identified in requests via URIs in RESTful webservices, e.g. the server could send data as XML or JSON.
  - Resource manipulation through representations: when a clients has a representation of a resource, it has enough information to work with the resource's state.
  - Self-descriptive messages: each message includes enough information to describe how to process it.
  - Hypermedia as the engine of application state: starting from a base URI, a client can dynamically access all resources it needs. I.e., starting from a homepage a client can access resources further down the hierarchy.
- Optional - Code on demand: servers can temporarily alter client functionality by transferring executable code such as JavaScript.

[2]Uniform Resource Name

3.1.2. Design patterns

As the name suggests, design patterns provide a pattern for how the actual implementation should look. The difference between an architectural pattern and a design pattern is the level on which it is applied: architectural patterns always end up on top of design patterns.
Design patterns can roughly be divided into three main categories[GHJ+94][DSN03], namely behavioral patterns such as the observer, structural patterns such as adapter and proxy and creational patterns such as the abstract factory and singleton.

3.1.2.1. Behavioral patterns

Behavioral patterns focus not only on how objects should be described but also on how they communicate with one another. We can further distinguish between class patterns and object patterns. Class patterns depend on the use of inheritance to distribute behavior among classes, object patterns depend on object composition to define how objects work together in order to perform tasks. In what follows we will briefly go over some example patterns.

Name	Description
Mediator	The Mediator pattern is a way of defining an object that can be used to coordinate how a set of objects interacts. This promotes loose coupling between objects. It consists of a Mediator class that defines the interface which will be used, Colleague classes which communicate with the Mediator object, and a ConcreteMediator class which implements the coordination of Colleague's.
Observer	The Observer pattern defines a one-to-many relationship between objects so that when a change of state occurs all dependents are automatically updated. Using this patterns allows for encapsulating different aspects of an abstraction into their own class, making it possible to let them vary and reuse them independently. The pattern consists of four parts: the Subject, Observer, ConcreteSubject and ConcreteObserver. The Subject knows who its observers are and provides an interface for managing them, the ConcreteSubject stores the state which is relevant for the related ConcreteObserver objects and notifies them when the state changes. The Observer defines the interface used for updating objects when notified by a subject, the ConcreteObserver implements this interface and maintains a link to its ConcreteSubject object to keep its state synced.
State	The State pattern allows an object to change its behavior based on the state it is in at run-time. Using this pattern allows each branch of a conditional statement to be encapsulated in its own class, allowing states to be treated as proper objects which can vary independently from other objects. This pattern consists of three parts: a Context, State and ConcreteState subclasses. The Context defines the interface which will be available for clients and maintains an instance of the ConcreteState subclass that represents the current state. The State class defines the interface for encapsulating the behavior associated with a particular Context state. Each of the ConcreteState subclasses implements the behavior associated with that state.

3.1.2.2. Structural patterns

Structural patterns detail how classes and objects should be composed to form larger structures. Structural class patterns use inheritance to compose interfaces, which can be useful for making libraries work together or conforming interfaces. Structural object patterns on the other hand detail how to compose objects to realize new functionality. In what follows we will briefly go over some example patterns.

Name	Description
Adapter	The Adapter takes an interface and transforms it into another interface a client expects, i.e. it resolves compatibility issues. It is both a class and object pattern in which an Adapter takes the interface of an Adaptee and adapts such that it conforms to the interface of the Client.
Bridge	The Bridge pattern allows for an abstraction to be decoupled from its implementation so they can vary independently, meaning abstractions can be reused. This pattern consists of 4 classes: Abstraction, RefinedAbstraction, Implementor, ConcreteImplementor. The Abstraction defines an interface which defines higher level operations and maintains a reference to an object of the Implementor type, while the RefinedAbstraction class extends the interface as defined by Abstraction. The Implementor class declares an interface which is implemented by the ConcreteImplementor. This Implementor interface only provides basic operations which serve as basis for those of the Abstract class.
Composite	The Composite pattern composes objects into tree structures to represent part-whole hierarchies, allowing clients to treat compositions and individual objects uniformly. This pattern consists of an interface class Component as well as subclasses Leaf and Composite. The Component class implements default behavior for the interface common to all classes. The Leaf class represents leaf nodes, i.e. objects that have no children, and defines the operations which can be executed on base objects. The Composite class defines the behavior for components that can have children and maintains a list to all of its leaves.
Facade	A Facade is an object structural pattern which provides a unified interface to a set of interfaces in a subsystem. The Facade knows which part of the subsystem is responsible for handling certain requests and delegates client requests to the correct part of the subsystem. The subsystem classes themselves don't know the Facade exists.
Proxy	The Proxy pattern is an object structural pattern which provides proxies or placeholders to provide access control for other objects. The Proxy object maintains a reference to the real object in order to have access. It provides an interface to the outside world which is identical to that of the referred subject. We can further distinguish between remote proxy (handling requests), virtual proxy (cache information about the subject) and protection proxy (performs access control).

3.1.2.3. Creational patterns

Creational patterns help make a system independent of how its objects are created, composed, and represented. They work by encapsulating the knowledge about which concrete classes the system uses as well as hiding how instances of these classes are created. This means that the system in general only knows what is defined in the interface of the abstract class. In what follows we will briefly go over a few of the creational patterns.

Name	Description
Abstract factory	The abstract factory pattern provides an interface for creating families of related or dependent objects without specifying concrete classes. In this pattern we distinguish 5 classes: AbstractFactory, ConcreteFactory, AbstractProduct, ConcreteProduct and Client. The AbstractFactory declares an interface for operations to create abstract product objects, the ConcreteFactory then implements these operations to create concrete product objects. The AbstractProduct class declares an interface for a type of product objects while the ConcreteProduct class implements this interface in order to define a concrete product object to be created by the corresponding factory. The Client only uses the interfaces as declared by AbstractFactory and AbstractProduct.
Factory method	A factory provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. This means that we have a class Product, which defines the interface, as well as the ConcreteProduct class which implements this interface. The Creator class declares the factory method, which returns an object of type Product, and may also define a default implementation of this method that returns a default ConcreteProduct object. The final class in the pattern is the ConcreteCreator which overrides the factory method from the Creator class to return a ConcreteProduct object. In this pattern, the Creator class relies on its derived classes to properly alter the factory method such that it returns the correct ConcreteProduct.
Singleton	The singleton pattern ensures there is only one instance of the class as well as provide a global access point. This is done by making the constructor a protected method in the class and declare a pointer to an instance as private member. This member is then accessed via the public Instance() method.

3.2. Normalised Systems Theory

Normalised Systems Theory, NST from here on, is an architecture that aims to provide software that is stable, maintainable and extendable.
In their methodology, the implementation process is viewed as a transformation I of functional requirements R into software primitives S such that $S = I(R)$. The transformation R can be separated into two categories, namely static transformation and dynamic transformation.
The **static transformation** defines how we can transform functional requirements into software primitives. To this end three requirements, as well as the corresponding transformations, are formulated

- Requirement 1

An information system needs to be able to represent instances of data entities D_m. A data entity consists out of a number of data fields $\{a_i\}$ which may be a basic data field representing a value, or a reference to another data entity.
$\longrightarrow$ Transformation implementation
Every data entity D_m is transformed into a data structure $S_m = I(D_m)$. This means that each data entity is instantiated as a software construct for data as provided by the programming language.

- Requirement 2
  An information system needs to be able to execute processing actions P_n on instances of data entities. A processing action consists of a number of consecutive tasks $\{t_J\}$. Such a task may be a basic task such as a unit of processing that can change independently or an invocation of another processing action.
  $\longrightarrow$ Transformation implementation
  Every processing action P_n is transformed into a processing function $F_n = I(P_n)$. This means that each processing action is instantiated as a software construct for processing as provided by the programming language.

- Requirement 3
  An information system needs to be able to input or output values of instances of data entities through connectors C_l.
  $\longrightarrow$ Transformation implementation
  Every I/O connector C_p of a data structure is transformed into a processing function $F_p = I(C_p)$ of the programming language, in the same way as a processing action, and makes use of the standard I/O functionalities of the language.

A software system thus becomes a set of software primitives S, consisting of a subset of data structures $\{S_m\}$ and a subset of related processing functions $\{F_n\}$.
The **dynamic transformation** ensures that evolving functional requirements can be caught. It does so by extending the static transformation with two additional requirements.

- An existing system representing a set of data entities $\{D_m\}$ needs to be able to represent both a new version of a data entity D_m that corresponds to including an additional data field a_i as well as a completely new data entity.

- An existing system providing a set of processing actions $\{P_n\}$ needs to be able to provide both a new version of a processing task t_j as well as an additional processing task and both a new version of a processing action P_n as well as an additional processing action.

3.2.1. Stability

To ensure stability in the software NST makes use of four design theorems: separation of concerns, data version transparency, action version transparency and separation of states.

3.2.1.1. Separation of concerns

Separation of Concerns is a theorem which is concerned with how tasks are implemented within processing functions. Taking into account that the goal of NST is to deliver software that is evolvable, we identify these tasks based on the concept of change drivers, i.e. a single concern within the application. In order to achieve stability, functions should not address more than one concern. This leads to the following formulation of the theorem.

Theorem 1 *A processing function can only contain a single task in order to achieve stability.*

This theorem can manifest itself in a number of ways.
A first manifestation is the use of an integration bus to manage communication between components and/or applications. Take for instance a classical application model where each component communicates directly with the others. This would mean that, for N components, there would need to be $\frac{N(N-1)}{2}$ connectors to facilitate communication. This form of communication is in violation with the theorem as it forbids the direct transformation between two protocols as such a transformer would be subject to more than one change driver. If we use a change bus we only need to add one single connector, which does not violate the theorem and is considerably better when taking stability into account. Using a change bus also reduces the amount of work for adding a new connector from $O(N)$ to $O(1)$.
A second manifestation is the use of external workflows as the workflow sequence, i.e. the sequence in which a number of actions are performed, is a separate change driver from the way each of the actions are implemented. Viewing workflows as separate change drivers allows us to modify the workflow, e.g. changing the order in which we want to execute certain steps, without changing the way in which the actions are implemented.
A final manifestation is related to the software architectures. The use of a multi-layer architecture, as described in section 3.1.1.2, separates each change driver into a separate layer.

3.2.1.2. Data version transparency

p278, 306

3.2.1.3. Action version transparency

p281, 307

3.2.1.4. Separation of states

p283, 308

3.2.2. Maintainability

In order to ensure maintainability of a software system, NST implements the notion of diagnosability in order to counter the inevitable entropy of the system. This diagnosability dictates that every observable system state is deterministically traceable to the software primitives causing the state.
This is vital as the number of internal states increases during each function invocation, leading to an increased complexity in the state space of the system. In this state space we can distinguish between internal states and external states.
Internal states describe the system during execution of an invocation, e.g. the steps you see when running a debugger in an IDE such as Intelij, and consist of the following items.

- The values of the global data structure, e.g. the values which become available when the software starts and can be used during the entire lifetime of the program. They only cease to exist when the program ends.

- The values of the local data structure which are only available during the lifespan of an individual invocation. They may contain intermediate results or store values obtain from external sources or other invocations.

External states on the other hand are far less detailed and contain only

- The external inputs the application may take during its lifetime.
- The external outputs the application may generate during its lifetime.

It goes without saying that the inputs will be taken into the internal state of the system and will reside there until the end of the application's lifetime, and that the outputs generated by the application will go to live on outside of the application and may remain there after the application has stopped.
If we want to use diagnostics to make statements about execution of software we need to define what is considered "properly executed". In NST this has been defined as "the uncertainty related to the identification of the cause of the notion 'has not been executed properly' " where the notion 'has not been executed properly' corresponds to the occurrence of an irregular execution of a task which caused improper execution or failure of the software.

3.2.3. Extendability

p315, 329, 452

3.2.4. Elements and layers

p363

3.3. Refactoring: fixing what's broken

Refactoring is the art of improving the quality of existing code without changing the application's appearance to the outside world. The goal of refactoring is to make code easier to understand, maintain, and modify, while reducing the risk of introducing new bugs and tackling technical debt.
The need for refactoring can arise due to various reasons. For instance, the code may have become difficult to understand or modify due to its complexity or lack of proper documentation. It may also have become outdated or redundant due to changes in business requirements or technological advancements. Some common refactoring techniques include identifying and isolating code smells, applying design patterns and principles, simplifying code and applying separation of concerns, and using tests to ensure correctness and consistency. Refactoring should be done incrementally and iteratively, focusing on small, manageable chunks of code that can be improved without causing major disruptions or breaking functionality.
Refactoring can be a challenging and time-consuming process, particularly for large and complex codebases. However, it should not be seen as a one-time task or a last-minute solution to fix code problems. Instead, it should be an integral part of the development process, focusing on continuous improvement, collaboration, and code quality.
Refactoring should not be confused with reengineering. Where refactoring is focused on improving the quality of existing code, the aim of reengineering is to alter the fundamental structure of a system or application. Refactoring can thus be used as a tool in reengineering efforts.

3.4. SonarQube

SonarQube is a tool that provides a detailed overview of numerous metrics when run on a codebase. In this thesis we will use SonarQube to obtain results on following metrics:

- Complexity: this value denotes the cyclomatic complexity of the codebase. A higher value indicates more branches in the codebase.
- Duplications: number of duplicated blocks of lines, files and percentage of duplicated lines. The expectancy is that these numbers will be relatively high when run on codebases following NST due to the way MicroRadiant generates its code.
- Code smells: the total count of code smell issues found in the codebase.
- Technical debt: provides an estimate for the effort to fix all code smells.
- Maintainability rating: rating given to the project relative to the value of the technical debt ratio. This takes into account the outstanding remediation cost, i.e. the sum of the estimated time to fix all open issues. If this time is
  - $\leq 5\%$ of the time that has already gone into the application, the rating is A
  - $6 \leq time \leq 10$ the rating is B
  - $11 \leq time \leq 20$ the rating is C
  - $21 \leq time \leq 50$ the rating is D
  - anything over 50% is an E
- Vulnerabilities: the number of vulnerability issues.
- Security rating:
  - A = 0 Vulnerabilities
  - B = at least 1 Minor Vulnerability
  - C = at least 1 Major Vulnerability
  - D = at least 1 Critical Vulnerability
  - E = at least 1 Blocker Vulnerability
- Security remediation effort: the effort to fix all vulnerability issues.

3.5. System under observation

In order to verify whether NST delivers on the promises it makes we decided to perform a small experiment.
We selected a small legacy system with as criteria that it should be written in Java and it should be a CRUD based system, basically eliminating games for example and music processing software. The system we chose was found on GitHub at https://github.com/VaibhavTyagi010/Online_CabBookingApp.

This Cab booking application was created using a specific kind of **multi-tier** architecture, namely Controller-Service-Repository. In this implementation the developers decided to extend this pattern with two additional layers, resulting in a total of five layers: Controller, Entity, Exception, Repository and Service.

3.5.1. Controller layer

Figure 3.4 provides a diagram of how the Control layer is implemented. As we can see the classes in the Control layer provide the CRUD, Create-Read-Update-Delete, operations needed for the application as well as the necessary getters and setters to allow access to the private members.
Classes in this layer define user-interface according to the REST principles. They relay requests they receive to the corresponding Service layer object.

3.5.2. Entity layer

Classes in the Entity layer define how data is stored in the database. Figure 3.5 provides an overview of the entities which are currently present in the application and how they are connected.
It is worth noting that in the current implementation, the use of inheritance breaks the Separation of Concerns principle.

3.5.3. Exception layer

3.5.4. Repository layer

3.5.5. Service layer

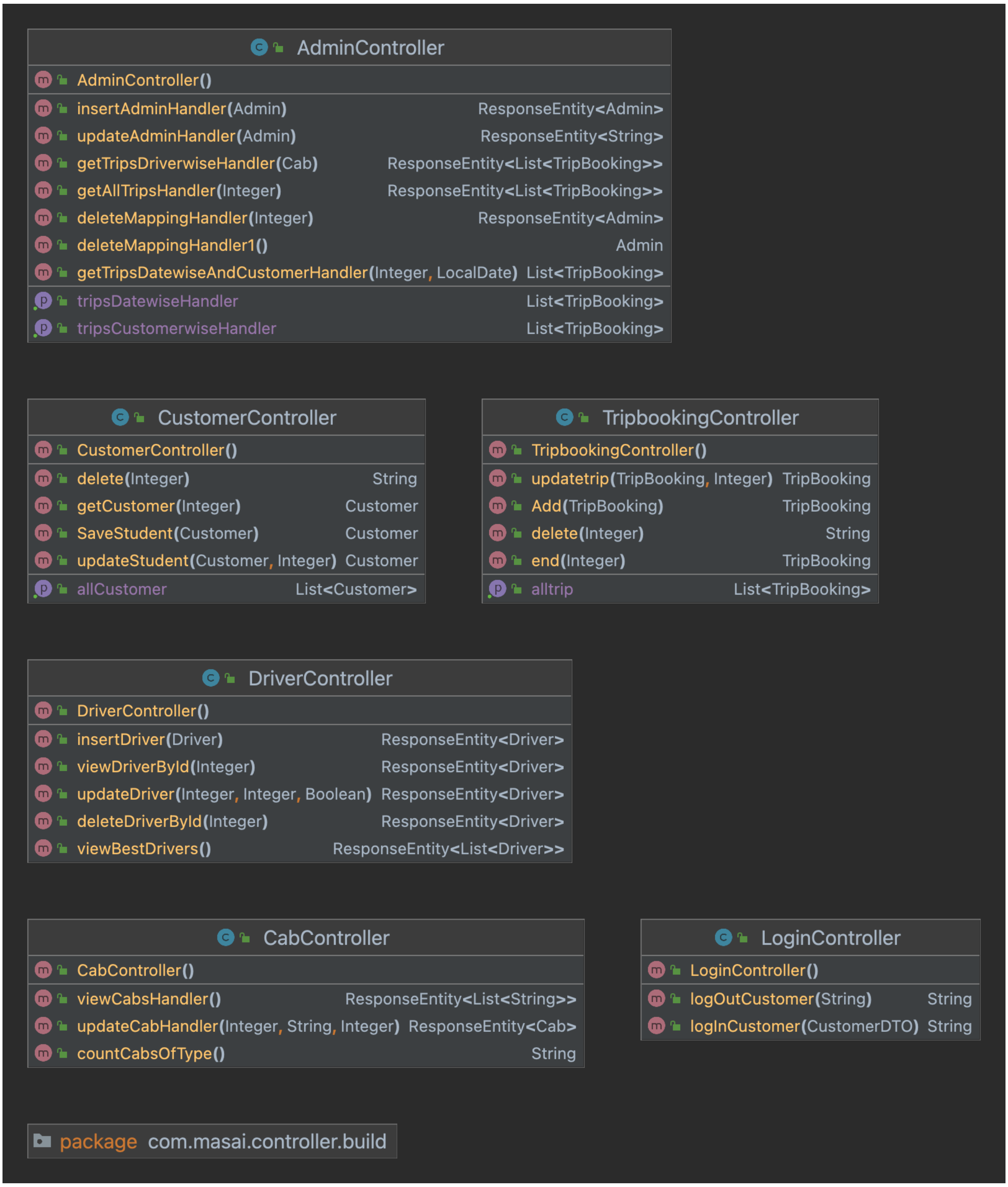

Figure 3.4.: Diagram of the control layer

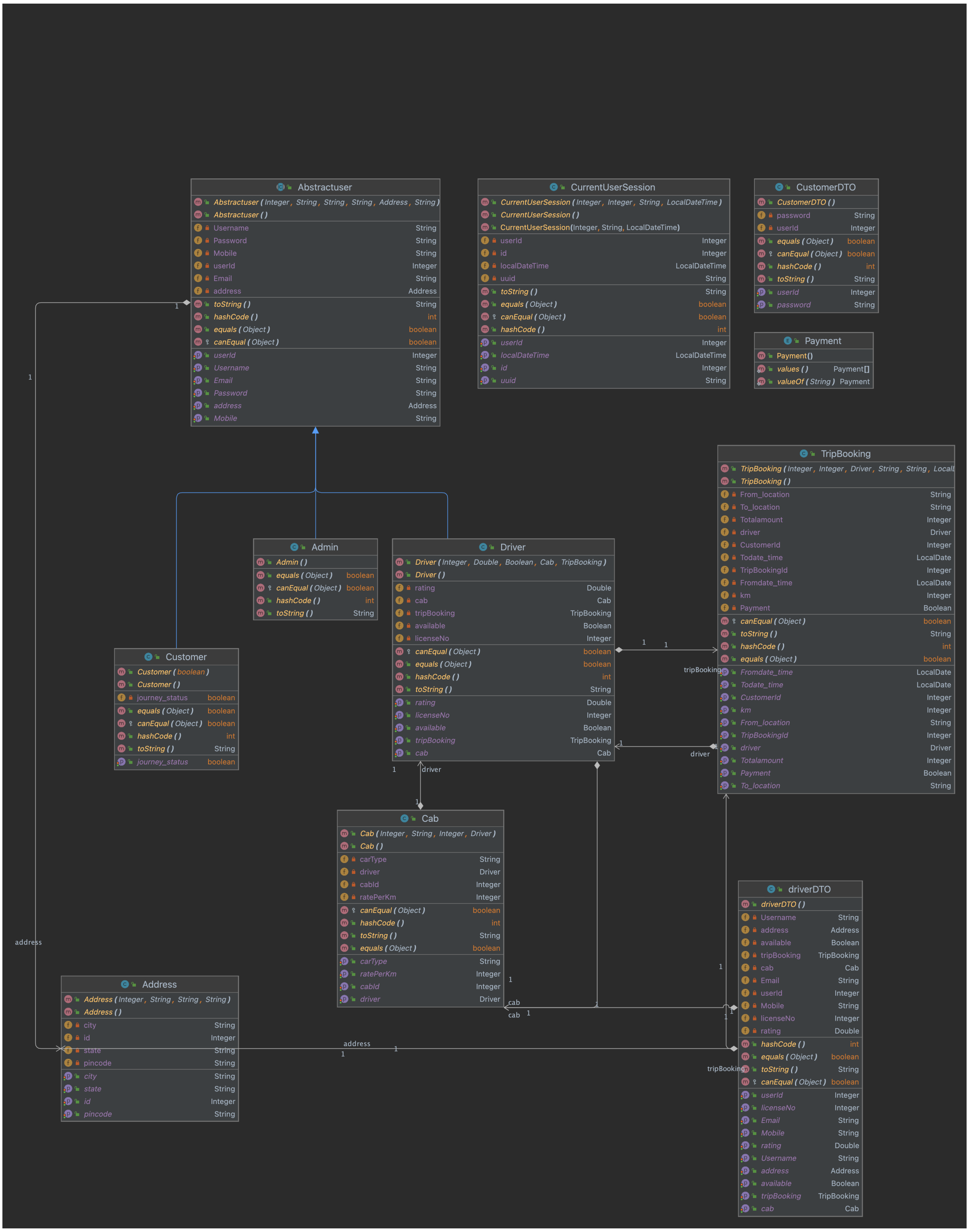

Figure 3.5.: Diagram of the entity layer

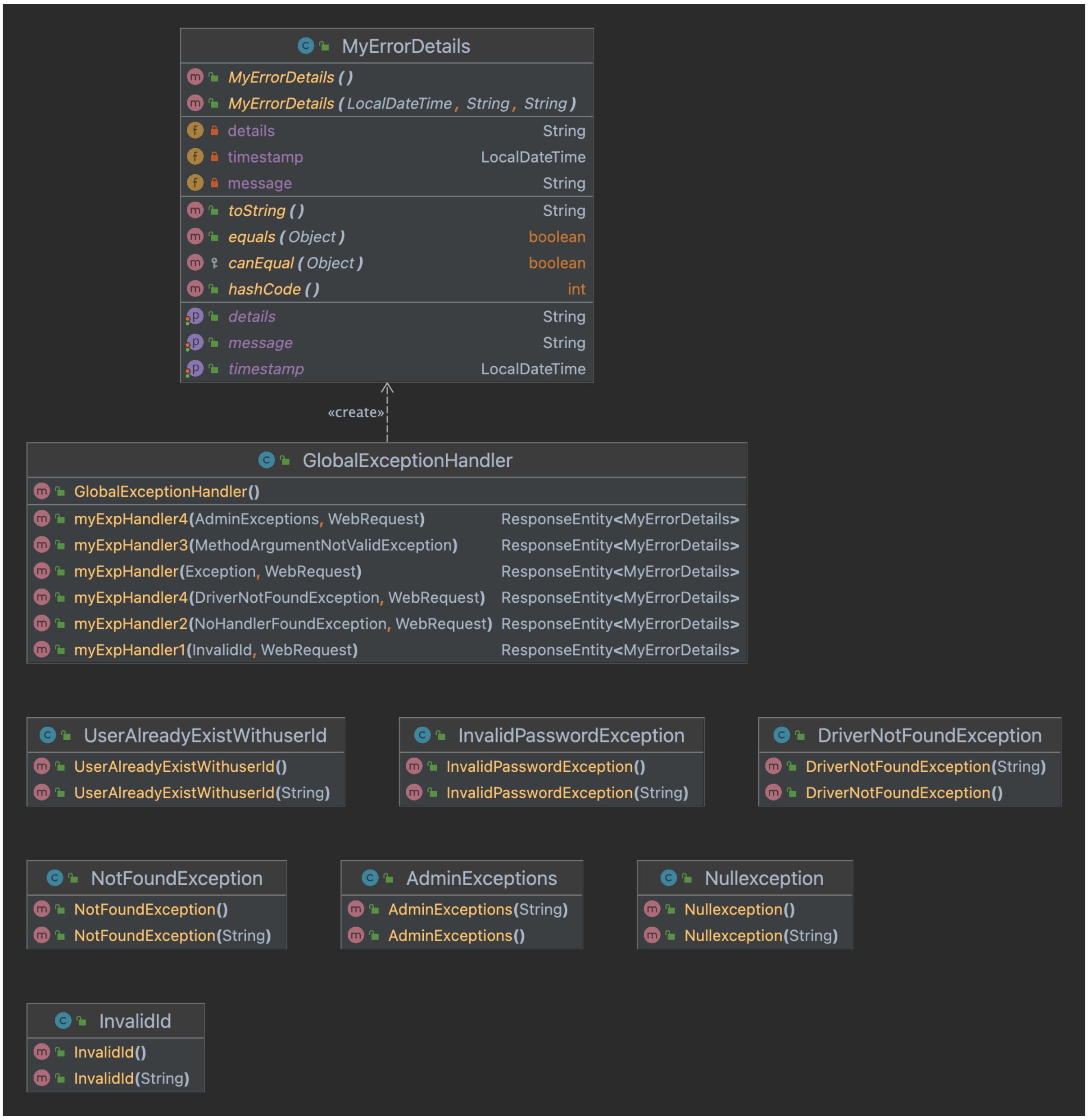

Figure 3.6.: Diagram of the exception layer

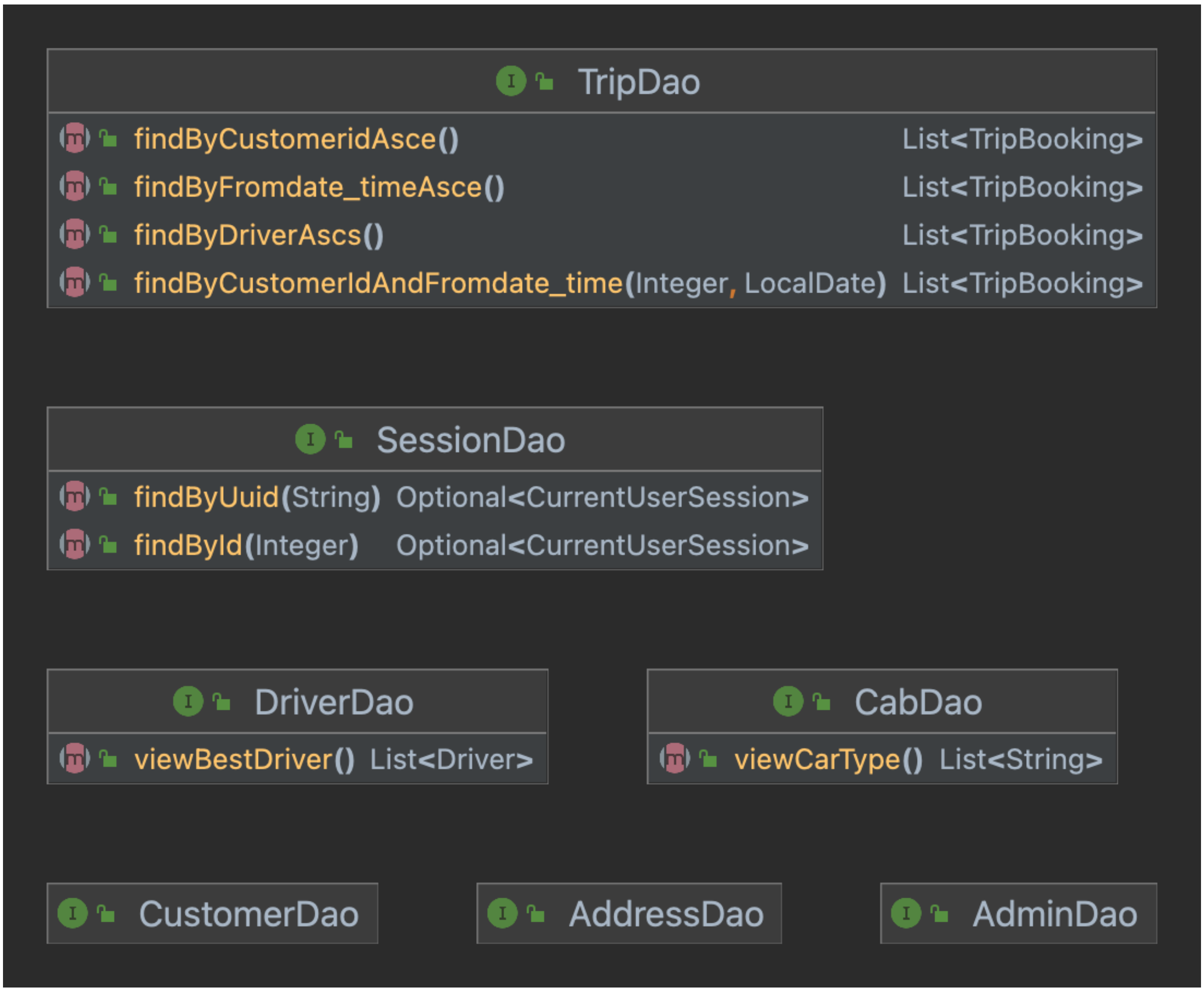

Figure 3.7.: Diagram of the repository layer

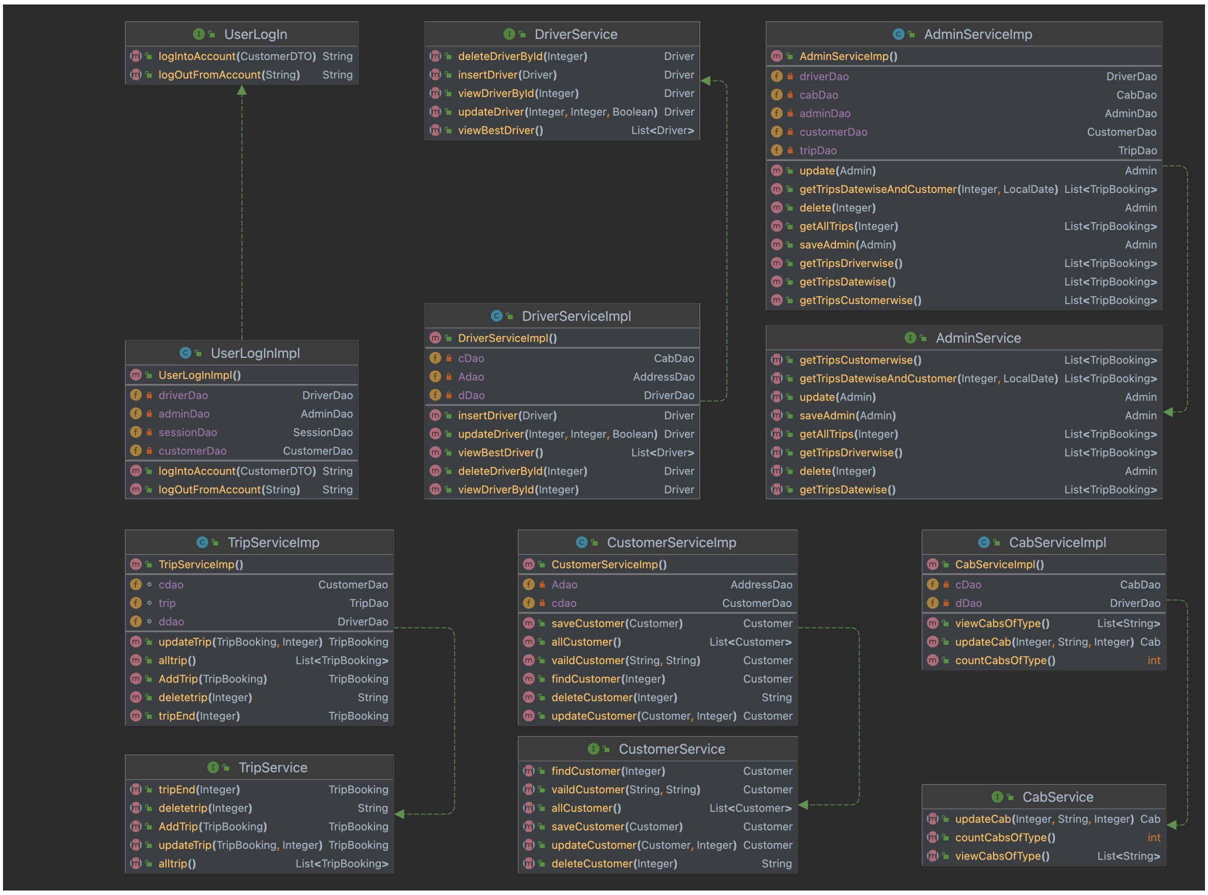

Figure 3.8.: Diagram of the service layer

4. Refactoring into NST

5. Conclusions

A. Bibliography

Bibliography

[Bab21] Shibu Babuchandran. Software release life cycle (srlc): Understand the 6 main stages, 2021. [Online; publication date Dec 6 2021]. 4

[DSN03] Serge Demeyer, Ducasse Stephane, and Oscar Nierstrasz. *Object-oriented reengineering patterns*. Morgan Kaufman Pub., 2003. 11

[ECPB12] Thomas Erl, Benjamin Carlyle, Cesare Pautasso, and Raj Balasubramanian. *SOA with REST Principles, Patterns Constraints for Building Enterprise Solutions with REST*. Pearson, 2012. 9

[Ext21] Extensibility. Extensibility — Wikipedia, the free encyclopedia, 2021. [Online]. 4

[GHJ+94] Erich Gamma Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Grady Booch. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. 11

[NGI] NGINX Glossary. What is load balancing? [Online]. 7

[Pie] Michael Pierce. Software release life cycle (srlc): Understand the 6 main stages. [Online; no publication date]. 3

[Ric15] Mark Richards. *Software Architecture Patterns Understanding Common Architecture Patterns and When to Use Them*. O'Reilly, 2015. 7

[sof] What is software rot, and why does it happen? [Online; no publication date or author]. 3

[Sof23] Software rot. Software rot — Wikipedia, the free encyclopedia, 2023. [Online]. 3

[sta21] The importance of stability and reliability testing in software development, 2021. [Online]. 3

[tec] What is technical debt and what should you do about it? [Online; no publication date or author]. 3