Academic year
2022 - 2023

Software Design
Normalised Systems Theory versus Refactoring

Beau De Clercq

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

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

University of Antwerp
Faculty of Science

Contents

Abstract

Software Design

Normalised Systems Theory versus Refactoring

Beau De Clercq

During software development developers aim to develop software that meets certain quality properties such as stability, maintainability and extensibility. While the theoretical side provides us with tools and principles with which we can obtain these properties, in practice they are not always achieved or maintained. In this thesis we will take a look at the feasibility of refactoring a Java legacy system so that it follows the principles of Normalised Systems Theory, which if followed promises to deliver software which is stable, maintainable and extensible.

Before we explain how we will measure in what ways Normalised Systems Theory affects the desired quality properties we will define some key concepts as well as the tools used. We start by defining what a software architecture is and what the difference is between architectural patterns, which are focused on the higher level of the global structure, and design patterns, which are focused on the lower level of classes. After these definitions we will dive into what Normalised Systems Theory is and how it aims to deliver stability, maintainability and extensibility followed by a small refresher on what refactoring is.

Once these concepts have been detailed we introduce the tools which were used to facilitate the comparisons. The first category are the tools used to determine the metrics, namely CodeScene and SonarQube. CodeScene is used to obtain a more global overview of what are called hotspots and how healthy the code in these hotspots is, proposing solutions to handle unhealthy zones. SonarQube on the other hand provides a more in-depth insight in the code by performing static code analysis, reporting various kinds of bugs and code smells together with how to resolve them. The second category of tools are the tools used to transform our Java system so it complies with Normalised Systems Theory, namely μRadiant. This tool is used to model an application and generate code which can then be extended on by a developer.

After the key concepts and tools we explain how our comparison was set up. In order to obtain meaningful results we used three systems. The first one is our starting project and was developed in the traditional way. Our second project is a transformation of our initial system into Normalised Systems Theory. The final project is our baseline Normalised Systems Theory project and is used to verify whether the results we obtained from our translation are in line with what can be expected from an industrial size system. In the following section we explain how we transformed our system, what the lifecycle of μRadiant looks like and some pointers on how to implement custom code. We conclude the chapter by providing an overview of the metrics we used to evaluate the quality properties and some features which we feel are currently missing in μRadiant.

The final chapter is dedicated to the actual comparison. In this chapter we provide overviews of the different metrics and provide suitable interpretations and explanations. At the end of the chapter we draw a general conclusion on whether or not it is feasible to use μRadiant and Normalised Systems Theory on an industrial level.

1. Introduction

In the current software landscape, properties such as maintainability, extensibility and stability are highly valued and necessary to avoid code rot and ensure software quality.[Sof23] To ensure this quality, developers use various techniques such as testing and adherence to coding best practices. By focusing on quality during the development process, software can be made more robust, less prone to errors, and easier to maintain and evolve.
Software evolution is an inevitable process that occurs throughout the software's lifecycle. As technology advances and user requirements change, software needs to be able to adapt and evolve accordingly. During software evolution, three key aspects become crucial: stability, maintainability, and extensibility. Stability refers to the ability of software to perform consistently without unexpected failures or disruptions. It ensures that the software remains reliable and meets the users' expectations. Maintainability is the ease with which software can be modified, debugged, and repaired. A maintainable system is such that it allows developers to efficiently make changes and fix issues without introducing new problems. Extensibility refers to the ease with which new functionalities can be added to software. An extensible system can be easily expanded without significant refactoring, meaning it is able to adapt to evolving user needs and technology. These three aspects - stability, maintainability, and extensibility - are crucial for the long-term success and viability of software as it evolves over time.

Achieving these quality attributes in software can be done using different approaches. A first approach would involve following coding best practices and the use of analyzing tools like SonarQube or CodeScene. Developers can follow coding standards, design patterns and best practices to write clean code which will make code easier to maintain and extend. Tools like SonarQube and CodeScene can be used to analyze the codebase, identify potential issues or code smells, and provide actions that can be taken.
Another approach to achieve these attributes is by using a model-driven development approach. In this type of approach we create abstract models which capture the system's requirements, behavior and structure which are then used as blueprints for generating the code. By using model-driven development tools, such as μRadiant for Normalised Systems Theory, developers can ensure consistency, reduce human error and improve the overall quality of the generated code. The models can be easily modified and extended to accommodate changes in requirements or new features, making them more extensible and the generated code will be more maintainable as it adheres to predefined patterns and standards. This type of approach can significantly increase stability, maintainability, and extensibility of a system.

In this thesis we will look at both methods and compare how they perform in obtaining the desired quality attributes. We will do so by first taking a closer look at a project that has been build using the traditional methods of coding according to patterns and best practices. To this end we have selected a small system with a clear structure and well defined roles. We will further describe this system and how we will evaluate it in section 5.1 and section 5.4.
Secondly we will reproduce this system using a model-driven approach via μRadiant which, as will be explained in section 4.3, is a tool that can be used to model a system in terms of components and data entities based on which code can be generated and rejuvenated. As we aim to compare how both system perform with respect to the desired quality attributes, we will evaluate this system as well using the metrics described in section 5.4.

2. Research question

The goal of this thesis is to look at the feasibility of refactoring a legacy system so they comply to the NST principles and verify that the stability, maintainability and the extensibility of the system have improved. To this end we will do the following

- 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.

The research question can thus be formulated as follows: If we take a legacy system and transform it so that it complies with the principles and guidelines defined by Normalised Systems Theory, will the promised benefits be visible immediately or will we only be able to tell a difference on the long run?

In what follows we make a few assumptions

- Refactoring a system ın such a way that it complies with the principles of 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 stability of the system.

3. Background

In this chapter we will define the concepts used in this thesis. These concepts are as follows

- Software Architectures: As Normalised Systems Theory defines an architecture, we will first give a brief overview of some relevant of architectures and patterns.
- Normalised Systems Theory: Normalised Systems Theory provides principles and guidelines which, if followed correctly, result in software which is stable, maintainable and extensible.
- Refactoring: The line between recreating an application, also known as reengineering, and refactoring is fairly thin. To better distinguish between them we will see how they are related to each other.
- SonarQube: As this will be the basis on which we will compare our systems, we introduce what SonarQube is and how we intent to use it.
- CodeScene: We will use CodeScene to obtain an alternative view on our comparison.
- μRadiant: The tool we used to model and generate code for our system so it complies to the principles of Normalised Systems Theory.

3.1. Software Architectures

Just as with the physical architecture, 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 and microservices.

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.

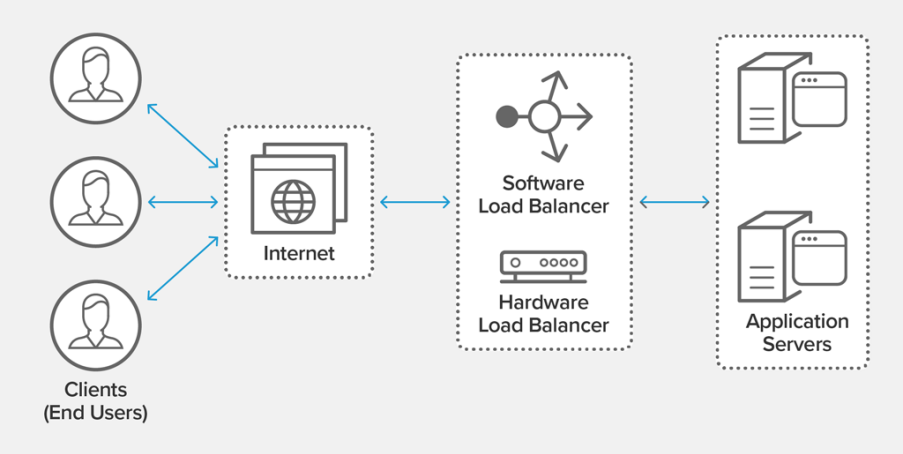

Figure 3.1.: Load balancing diagram

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. Model-View-Controller

As defined on Wikipedia, the model-view-controller pattern is an architectural pattern which can be used to develop interfaces by separating the application in three connected components.[Mod21]
The model component contains the internal data structures and manages both data and logic implementations for the application. The view component handles all visualisations and registers user input for the controller to process. The controller component takes the user input and processes it on the model component.

3.1.1.3. 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 is Presentation-Business-Persistance-Database.[Ric15] An example is shown in Figure 3.3 [Ric15] 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

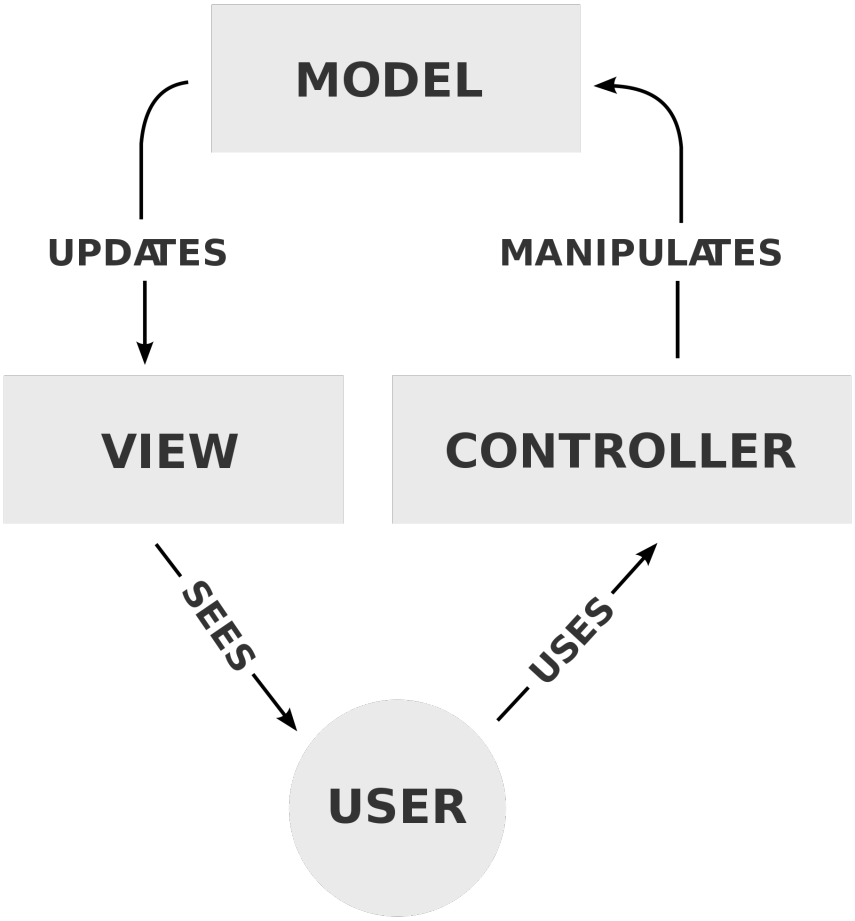

Figure 3.2.: Flow of the MVC pattern

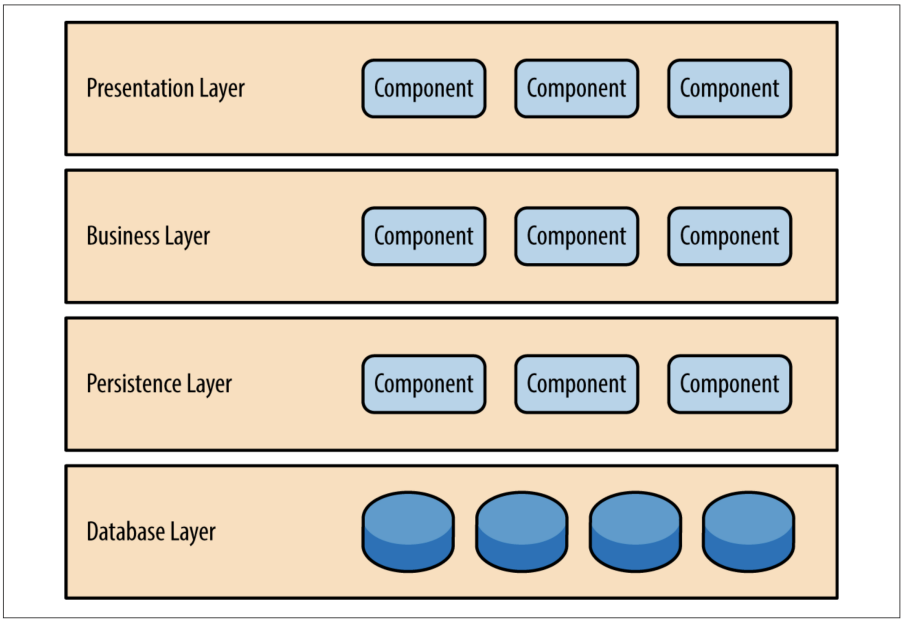

Figure 3.3.: Example of multi-tier architecture

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.

3.1.1.4. 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 3.4. In this example

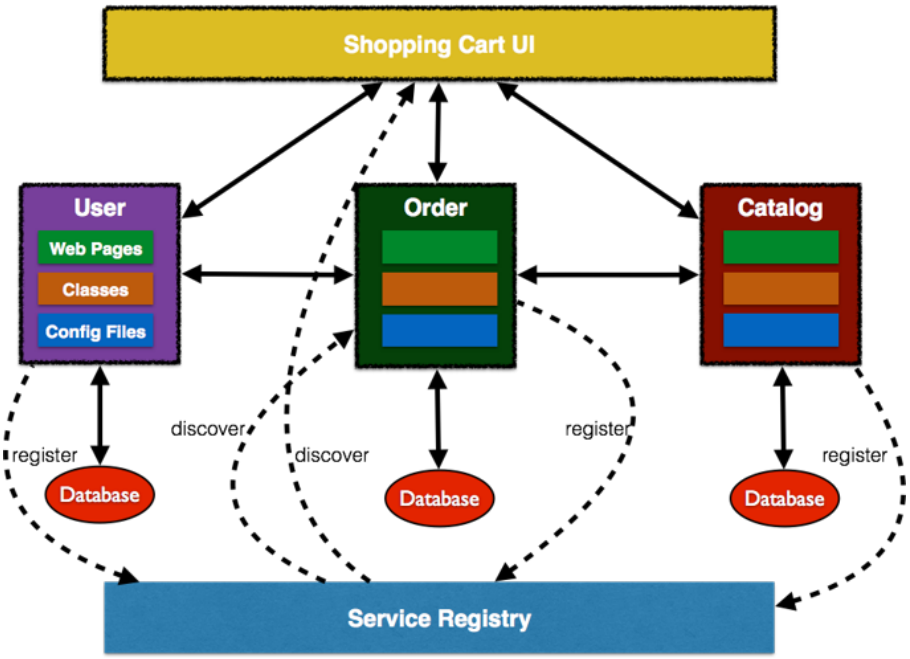

Figure 3.4.: Example of microservice architecture

our application consists 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.

3.1.2. 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.

As described in "Design Patterns Elements of Reusable Object-Oriented Software", design patterns can roughly be divided into three main categories, namely behavioral patterns such as the observer, structural patterns such as adapter and proxy and creational patterns such as the abstract factory and singleton.[GHJ+94][DSN03]

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
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.
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. 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

In what follows we will provide the essence of Normalised Systems Theory as written in "Normalised Systems Theory" by Herwig Manneart, Jan Verelst and Peter De Bruyn.

Normalised Systems Theory, NST from here on, is an architecture that aims to provide software that is stable, maintainable and extendable. In what follows we will summarize the theory as it is described in "Normalised Systems Theory".[VM03]

In their method, the implementation process is viewed as a transformation I of functional requirements R into software primitives S such that $S = I(R)$. The transformation I 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 3.1.1.3, separates each change driver into a separate layer.

3.2.1.2. Data version transparency

The data version transparency theorem is concerned with how data entities are passed to processing functions. An entity has data version transparency if it can have multiple versions without affecting related processing functions: it should be possible to upgrade an entity without affecting related functions and methods that process it.

Theorem 2 *A data structure that is passed through the interface of a processing function needs to exhibit version transparency in order to achieve stability.*

This feature is in fact present in nearly every environment as this corresponds to the notion of polymorphism. Other notable examples include web services or microservices as detailed in section 3.1.1.4 and the concept of information hiding and encapsulation.

3.2.1.3. Action version transparency

This theorem is concerned with how functions are called by other functions. It means that functions can have multiple versions without affecting other functions that call it during their own execution, i.e. it should be possible to upgrade functions without affecting the rest of the system.

Theorem 3 *A processing function that is called by another processing function, needs to exhibit version transparency in order to achieve stability.*

In practice this theorem is present in concepts as polymorphism and the use of wrapper functions.

3.2.1.4. Separation of states

p283, 308
The Separation of States theorem is concerned with how calls between processing functions are handled. What this theorem tells us it that, keeping in mind evolvability of the system, we need a separate data structure to store error states which may arise at runtime. For example, assume we have a function that performs a certain retrieval operation. If we change the implementation of this function its interface

will remain the same since we need to comply with Data Version Transparency and Action Version Transparency. This new implementation may however lead to a new error state which requires certain handling. As the number of calls to this retrieval function may keep growing, so too will the number of error states grow. If we then have a separate data structure to handle error states we can simply trigger the corresponding action from the data structure irrespective of what the calling function was, i.e. we don't need to provide endless try-catch clauses. This leads us to the following formulation of the theorem

Theorem 4 *Calling a processing function within another processing function, needs to exhibit state keeping in order to achieve stability.*

This theorem mainly manifests itself in asynchronous communication and processing. A more relevant manifestation are workflow systems as the principle of Separation of States corresponds to stateful workflow where intermediate states from the workflow are stored internally. Think of this as a state machine where the presence of a certain state will trigger the execution of the processing function of that state.

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, in this context defined as "the degree of complexity, disorder or uncertainty present in a 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. Extensibility

To ensure extendibility in a system, NST focuses on modularisation and how to keep modularity combinatorics under control. In this section we will briefly discuss the law of exponential variation gains and law of exponential ripple cost as well as their impact on extendibility.

In what follows we will use following notations and conventions

- U_i: The unit of work in a system or module which cannot be used or activated separately.
- M_i: Module in a system or artifact which can be used or activated separately.
- k_i: Number of variants or versions of a specific unit of work or module i.
- d_i: Number of other modules that are dependent on module i in a system.
- c_i: Number of copies of a specific uniet of work i in other modules.

3.2.3.1. Cohesion: Law of exponential variation gains

Consider a system which consists of n units of work U_i. In general the need often arises for different variants or versions of a unit of work. Suppose our system contains N different units of work and that every such unit U_i exists in k_i different versions or variants, then we have a few modular strategies to cope with these variations.

We could opt to go with one global monolithic system module in which all versions of all units of work would coexist. Having k_i variants, this single module would contain

$$\sum_{i=1}^{N} k_i$$

system part versions. This module would also contain all the logic glue to keep everything together and choose which version to use. If we then want to introduce a new version of any unit of work, we need at least

$$\sum_{i=1}^{N} (k_i - 1)$$

upgrades for the overal module. It is clear that this way of working would lead to a rather complicated monolithic module and should be avoided.

As alternative option we could choose to work with a different variant or version of the single global module for every specific combination of variants for the various units of work. In case all possible combinations of variants would be required and provided, this would lead to

$$\prod_{i=1}^{N} k_i$$

different variants of the aggregation module. If every possible combination of variants would be provided, every unit of work i would be duplicated

$$\prod_{j=1}^{i1} k_j \cdot \prod_{j=i+1}^{N} k_j$$

times. This also implies that any change on unit of work i would imply the duplication of that change as many times as there are duplications of that unit of work.
Of course in practice not all possible combinations will be required nor will they be provided, but it is clear that the efforts for development and maintenance for such a system would be exponentially dependent on the number of individual variants.

The more elegant - and efficient - solution would be to use separate and cohesive module variants. Suppose the overal system is decomposed into N independent modules where every unit of work U_i is embedded in a separate cohesive module. This would mean that there is a separate module variant for every variant of every unit of work and that every unit U_i has k_i module variants or versions corresponding to the k_i variants of that unit. The total amount of module versions to develop and maintain - the effort - would then become

$$\sum_{i=1}^{N} k_i$$

while the total amount of possible system aggregations - the yield - would equal

$$\prod_{i=1}^{N} k_i$$

This means that there is an exponential relationship between the effort for individual module variants and the total yield of available system variations

$$Nk \longrightarrow k^N$$

In other words, the development and maintenance of various variants of modules leads to an exponential gain in possible system variations. We call this relation the law of exponential variation gains.

3.2.3.2. Coupling: Law of exponential ripple costs

We can distinguish three types of coupling.

We can have coupling between system modules. Suppose we have N different modules M_i which each contain a single unit of work U_i where every module M_i has k_i different variants and for each module M_i there are d_i other modules M_j that depend on it. Consider the case where we want to change a specific variant of a specific module. In case the module boundary or interface is not correctly encapsulated or some other dependency is not correctly shielded, it may be possible that these changes will ripple through to the related modules. The number of changes that might be needed would then equal

$$\sum_{j=1}^{d_i} k_j$$

where k_j is the number of variants of a dependent module M_j. It is even possible that changes will continue rippling to the dependent modules of the dependent modules, and when no guarantee would exists that the ripple effect would stop there it could lead to a series of changes

$$dk + d^2k + d^3k + \ldots$$

We call this exponential increase in efforts or costs the law of exponential ripple costs.

We can also have coupling between various systems. Suppose a module M_i is reused in another system as module M_x with version v_x and d_x dependent modules. Within this system the number of first order dependencies would then be

$$\sum_{j=1}^{d_x} k_j$$

But as mentioned before we could also have second, third, ... order dependencies within this system and many more in still more other systems. It is obvious that this calls for standardization of basic modules in various domains.

A final sort of coupling we can distinguish is coupling within system modules. Suppose a part of a module M_i is duplicated or copied in the k_i versions of this module. Changing this part would then imply k_i changes and, as this part is combined with the rest of module within that module, it is quite likely this change may ripple through the rest of the module leading to k_i identical changes and k_i different changes. This means that a common part in all versions of a specific module will lead to

$$2k_i$$

changes from which $k_i + 1$ are different. In case a unit of work is duplicated across C different modules, the number of changes could be even higher.

3.2.4. Elements and layers

Implementing the theorems as specified in section 3.2.1 yields an architecture in which 5 types of elements are implemented across 6 layers. For sake of clarity we provide an overview of the data elements in Table3.2.4 and an overview of the layers in Table3.2.4.

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.[Fow19]

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.[DSN03]

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.[Fow19]

Element	Description
Data	A Data Element is at the core of NS systems and represents information used in the application. Data Elements are often related to nouns used to describe the workings of your application.
Task	A Task Element corresponds to a 'verb'. It operates on a Data Element and will execute some actions. A Task will take the element it is working on and return a task result that can either be success or failure. It is important that tasks are implemented independent of the state of the Data Element it is working on.
Flow	A Flow Element describes a process in the application. It allows a user to define a number of states and transitions which describe the flow of this process.
Connector	Connector Elements provide I/O support and are provided as-is.
Trigger	The Trigger Element provides the functionality of triggering or periodically invoking the flow orchestrators and correspond to a system clock. It is almost completely confined to the Logic Layer.

Table 3.1.: Overview of the Elements in NST

Layer	Description
Data	The Data Layer contains all data and CRUD -Create Read Update Delete - classes to provide data access.
Control	The Control Layer contains the server-side logic, it receives requests from the View Layer and uses both the Shared and Proxy Layer to handle them.
Logic	Contains Bean classes which provide transactions and implementations of tasks and commands.
Shared	The Shared Layer is used by all other layers (except the View Layer) and contains classes which are mainly used by the other layers to communicate.
Proxy	The Proxy Layer provides encapsulation for the remoting technology. Artifacts in the Logic Layer use EJB to implement transactions and scaling, but they can only be accessed by retrieving the objects through JNDI lookups.
View	The View Layer contains the visualization of the application and defines the pages the user will see and interact with.

Table 3.2.: Overview of the Layers in NST

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.[Fow19]

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.

4. Tools for comparison

In this chapter we will introduce the tools we used in this thesis: SonarQube, CodeScene and μRadiant.

4.1. Tools for measuring quality

4.1.1. 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 μRadiant 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 ≤ 10 the rating is B
  - $11 \leq$ time ≤ 20 the rating is C
  - $21 \leq$ time ≤ 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.

4.1.2. CodeScene

Another tool that might result in useful results would be CodeScene, which is a tool based on dynamic analysis unlike SonarQube which only performs static code analysis. When used in a professional setting, CodeScene allows for developers to track how much time has been spent on certain parts of the code, so called hotspots, as well as define certain goals on problematic areas.

In this thesis we will only use the Hotspots and Code Health features from the hotspots view. This view allows us to quickly analyze our system and dive into modules/files with low health so a suitable plan of action can be developed. The other features of this view, such as Costs and Code Churn, and the other views in general, are currently not useful as they depend on previous commits to assess changes or require more in depth configurations.

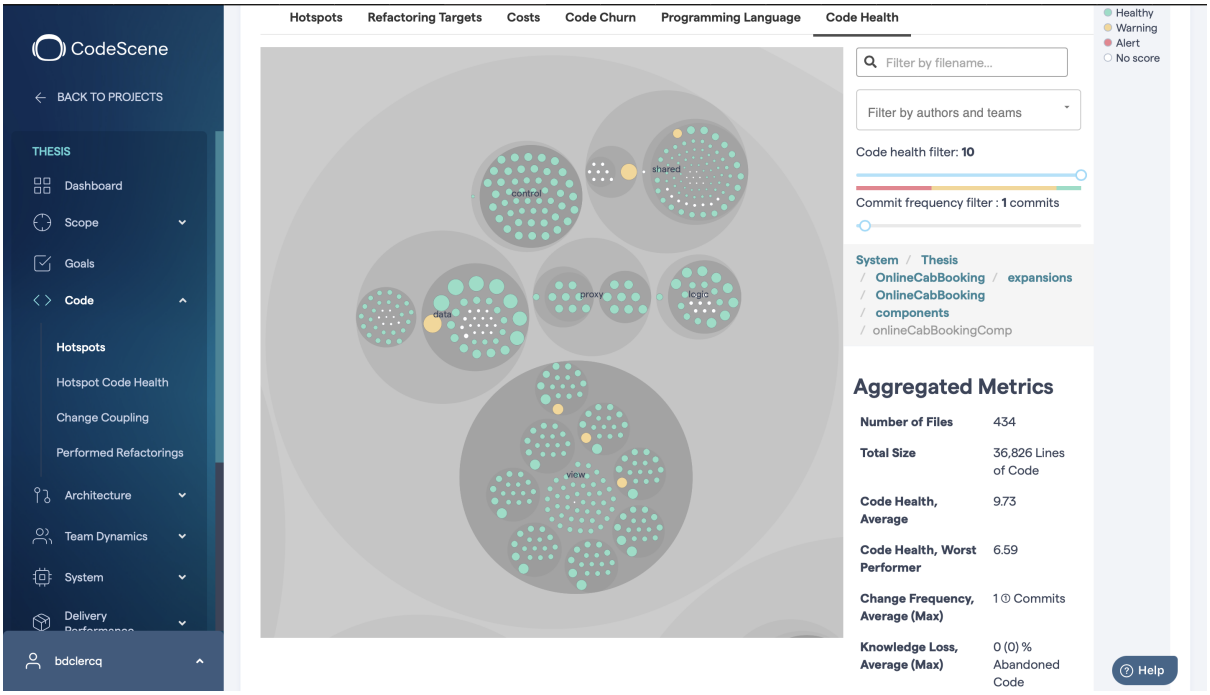

Figure 4.1.: Code health of the custom component according to Codescene

4.2. Quality properties

We will evaluate our systems based on three quality properties: stability, maintainability and extensibility.

4.2.1. 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.

Other important aspects of software stability are design and architecture.[Mar18] 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.[VM03] Additionally, adhering to best coding practices, such as error handling, exception handling, and defensive programming, can help prevent software instability caused by unexpected situations.

4.2.2. Maintainability

> **This characteristic represents the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements. - ISO 25010**

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. Take for instance the Agile development process, which has become increasingly popular over the past few years, in which we have a near continuous cycle of maintaining, updating and upgrading.[Pie] 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 is a term that refers to the cost you will pay in the future by trying to realise a quick fix in the present.[tecb][sof] 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.

4.2.3. Extensibility

Extensibility 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.[Ext21][Bab21] 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.

4.3. μRadiant

μRadiant is a tool which enables developers to model their NS application. After the modelling , μRadiant will generate a system which combines the microservice and multi-layer architectural patterns: the generated system will consists of multiple components where each component will consist of multiple layers. When we look at the layers separately, we see that most of them follow the principle of defining an interface and implementing it in a separate class. Exceptions to this are the Proxy Layer, which is implemented according to the proxy pattern, and the View Layer, which is implemented in HTML and JavaScript.
After generation of the system, μRadiant allows for immediate deployment using Docker, effectively eliminating the need for the use of the client-server pattern as described in section 3.1.1.1. As shown in Figure4.2, an NS Application has the following structure

- The application is represented by an Application element.
- Each Application can have multiple ApplicationInstances. An ApplicationInstance is a combination of settings to expand the application with.
- Each Application can have multiple Components, and Components can be reused among multiple applications. Components group Data, Task, Flow Elements and ValueFieldTypes.
  - DataElements represent information in the system, which can be stored, queried and updated.
  - TaskElements represent executable pieces of logic. A task has a target DataElement. When executed, the task receives an instance of that DataElement as parameter.
  - Flow Elements are linked to a DataElement with a dedicated status field. The Flow Element

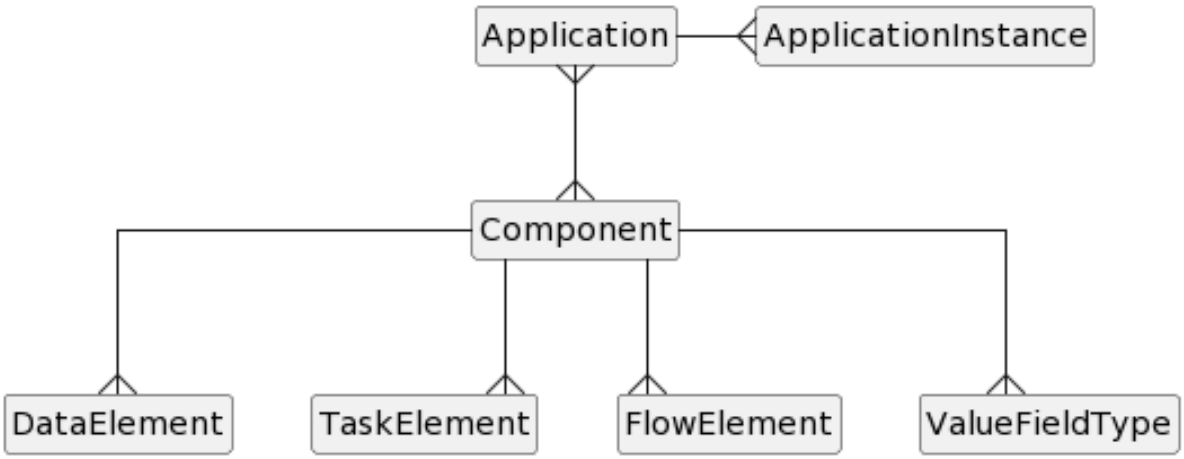

Figure 4.2.: Abstract structure of NS applications

describes a number of state transitions. Each transition has a begin-state and a TaskElement to execute if an instance of the DataElement reaches that state.

– ValueFieldTypes allow the implementation of custom types for ValueFields in the DataElements of the Component.

5. Comparison set-up

In what follows we will describe the practical details of our comparison. We will start by describing the system we chose as our starting point, providing an overview of the structure and some technical specifications. We will continue by providing an overview on how the transformation process works, what the μRadiant workflow is like and detailing how custom code can be injected in the generated skeleton. We will conclude by motivating how the evaluation metrics help assessing our quality properties.

5.1. System under observation

In order to verify whether NST delivers on the promises it makes we decided to perform a small comparison.

We selected a small legacy system with as criteria that it should be written in Java and it should be a CRUD - Create Read Update Delete - 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_Ca

5.1.1. Structure

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.

5.1.1.1. Controller layer

Figure 5.1 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.

5.1.1.2. Entity layer

Classes in the Entity layer define how data is stored in the database. Figure 5.2 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

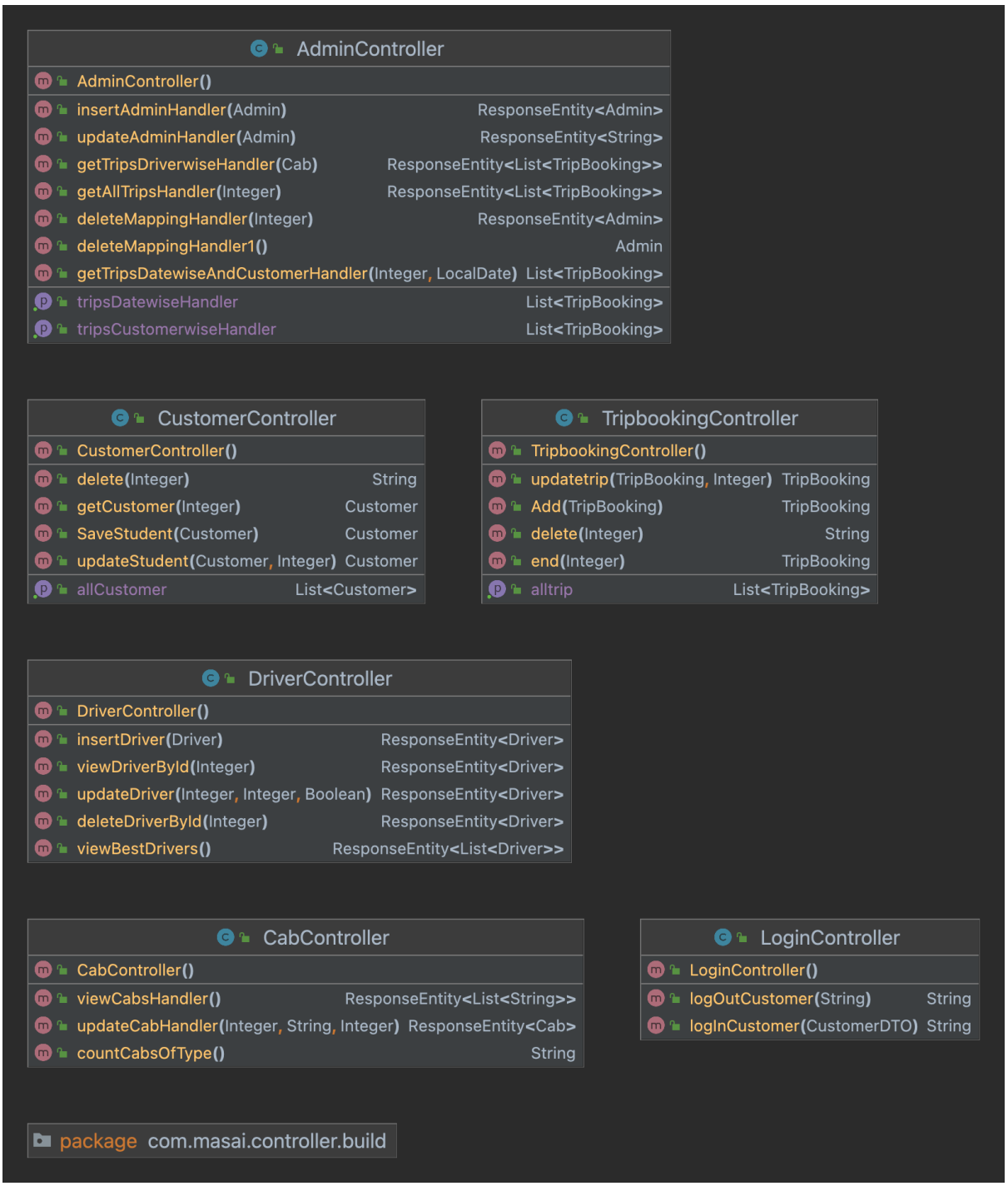

Figure 5.1.: Diagram of the control layer

Concerns principle. This can be circumvented by creating both the superclass and subclass as separate data elements and making the superclass entity a datamember of the subclass entity.

If we apply this to the current system, we would get data elements for Abstractuser, Customer, Admin and Driver where Customer, Admin and Driver would contain a pointer to an Abstractuser.

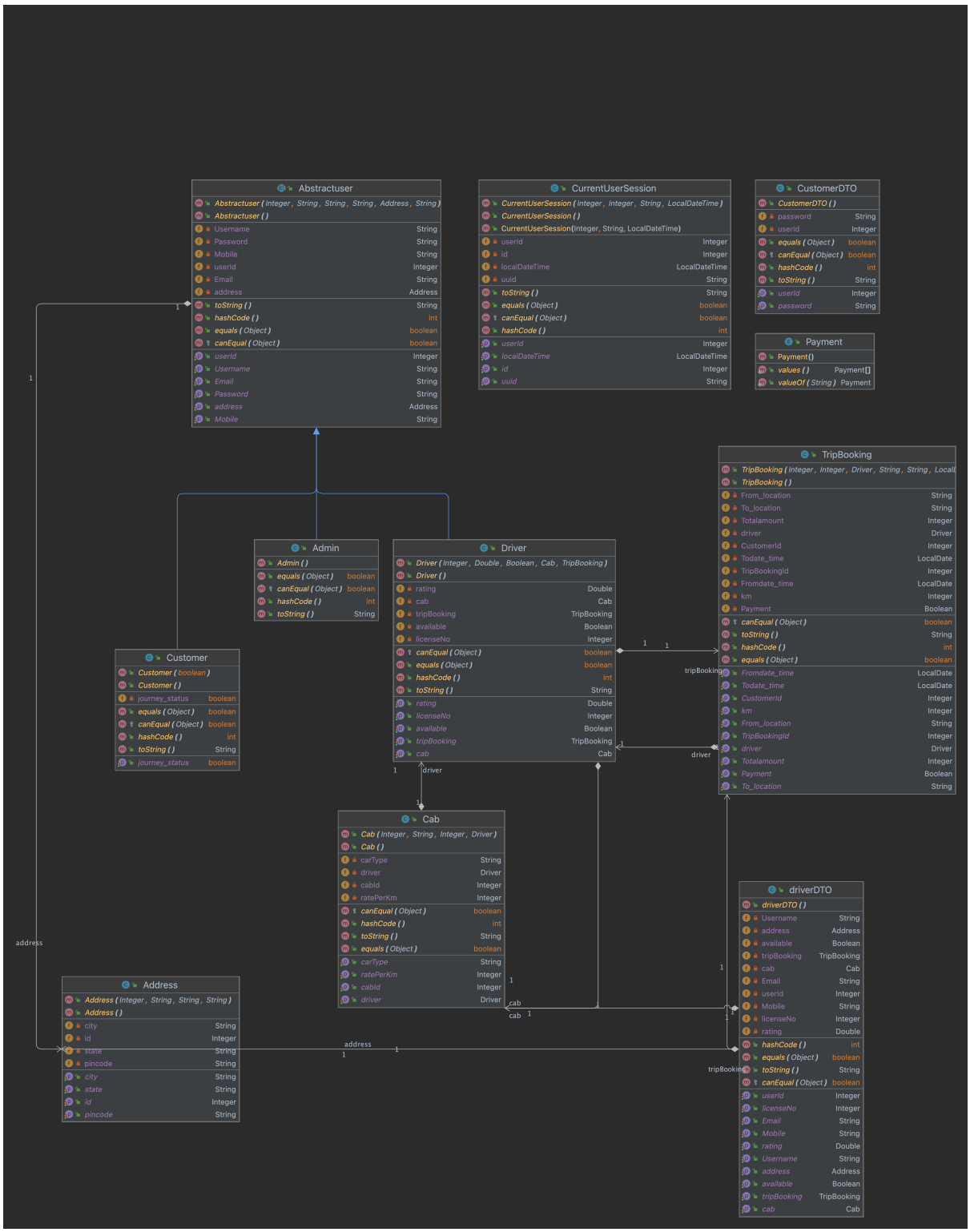

Figure 5.2.: Diagram of the entity layer

5.1.1.3. Exception layer

In Figure 5.3 we can see that some time has been put towards custom error handling.

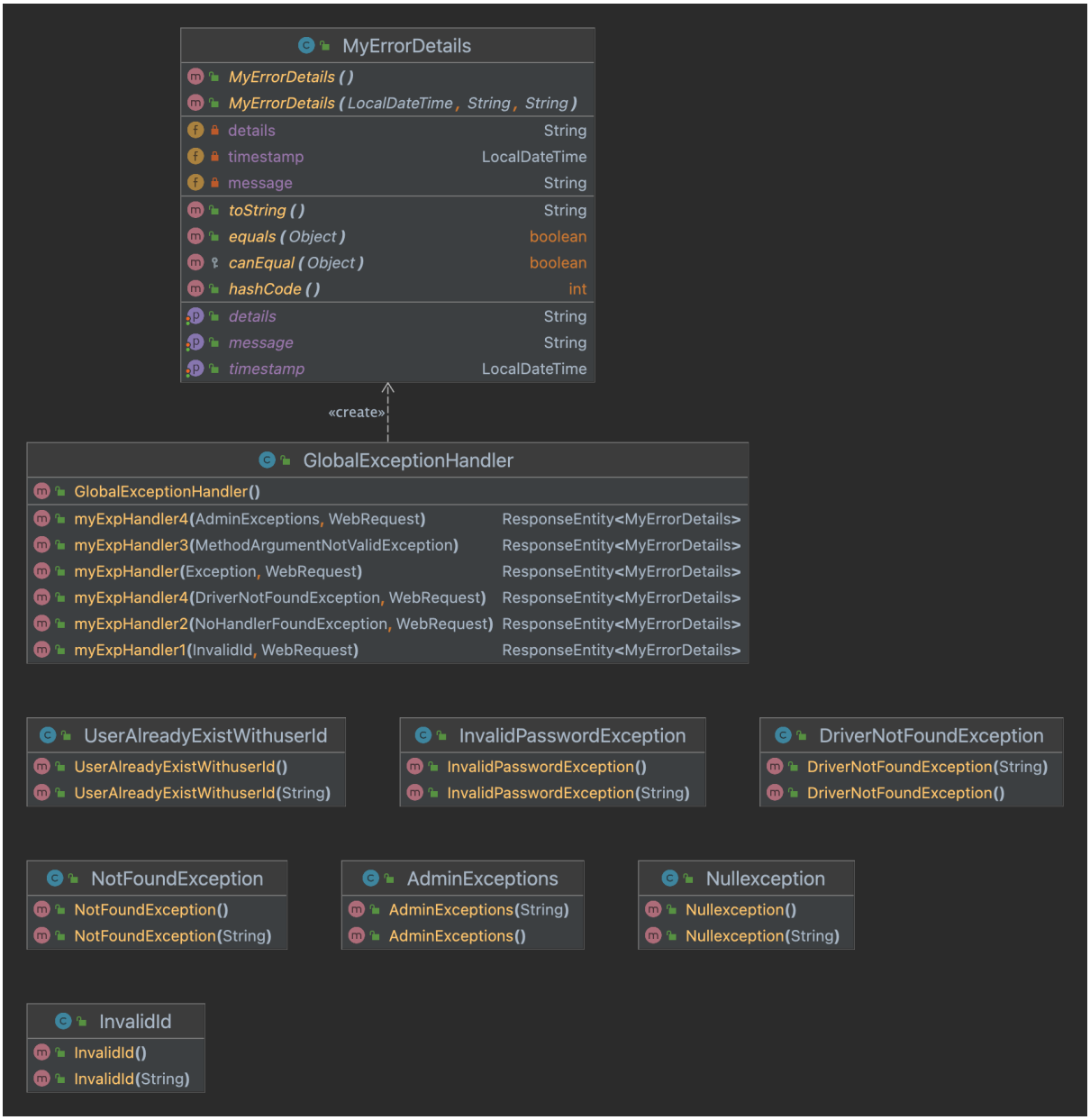

Figure 5.3.: Diagram of the exception layer

5.1.1.4. Repository layer

The DAOs in this layer, as given in Figure5.4, provide an API through which the application can access the database layer.

If we transform this system into NST, these finders and views can be generated simply by adding them to the corresponding data element in μRadiant.

5.1.1.5. Service layer

The service layer calls the API defined by the Repository layer to allow users to perform CRUD operations in the application. Figure5.5 provides an overview of the services which are currently provided to

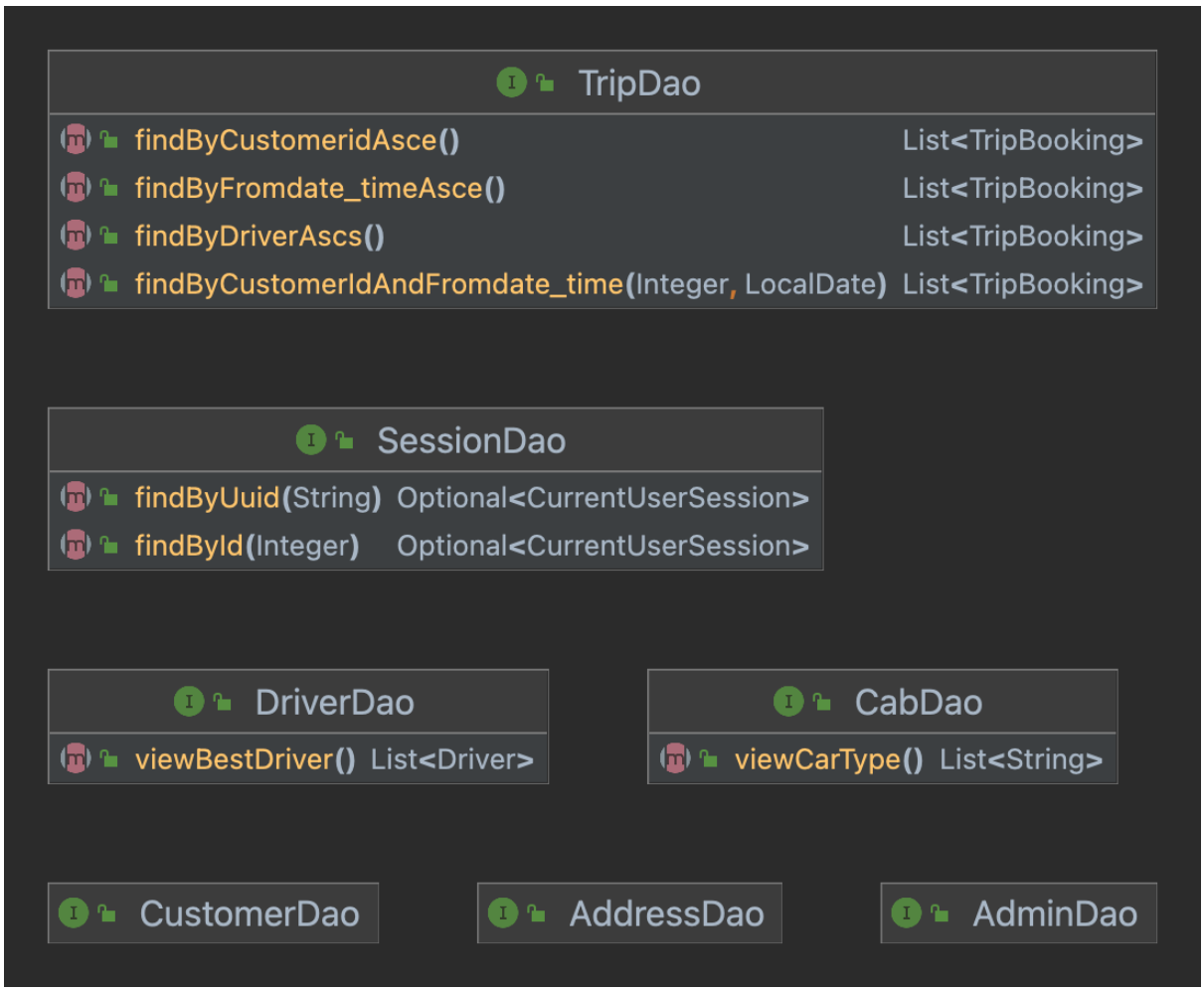

Figure 5.4.: Diagram of the repository layer

the user.

5.1.2. System analysis

For this thesis we decided to go with a small project which was developed by a single original developer and consists mainly of CRUD operations. A technical overview is given in Table 5.1.2. As these technical specs show this project leans itself for use in a comparison study: it is small enough so a single developer can learn to work with it in a very short amount of time while still being large enough to include sufficient functionality to form a basis for a system which, over time, may evolve into a industrial level application.

As we feel that it was not necessary to verify whether NST delivers on its promises as detailed in section 3.2, we decided to not use an industrial level system in this thesis to keep the overall complexity manageable and the work doable.

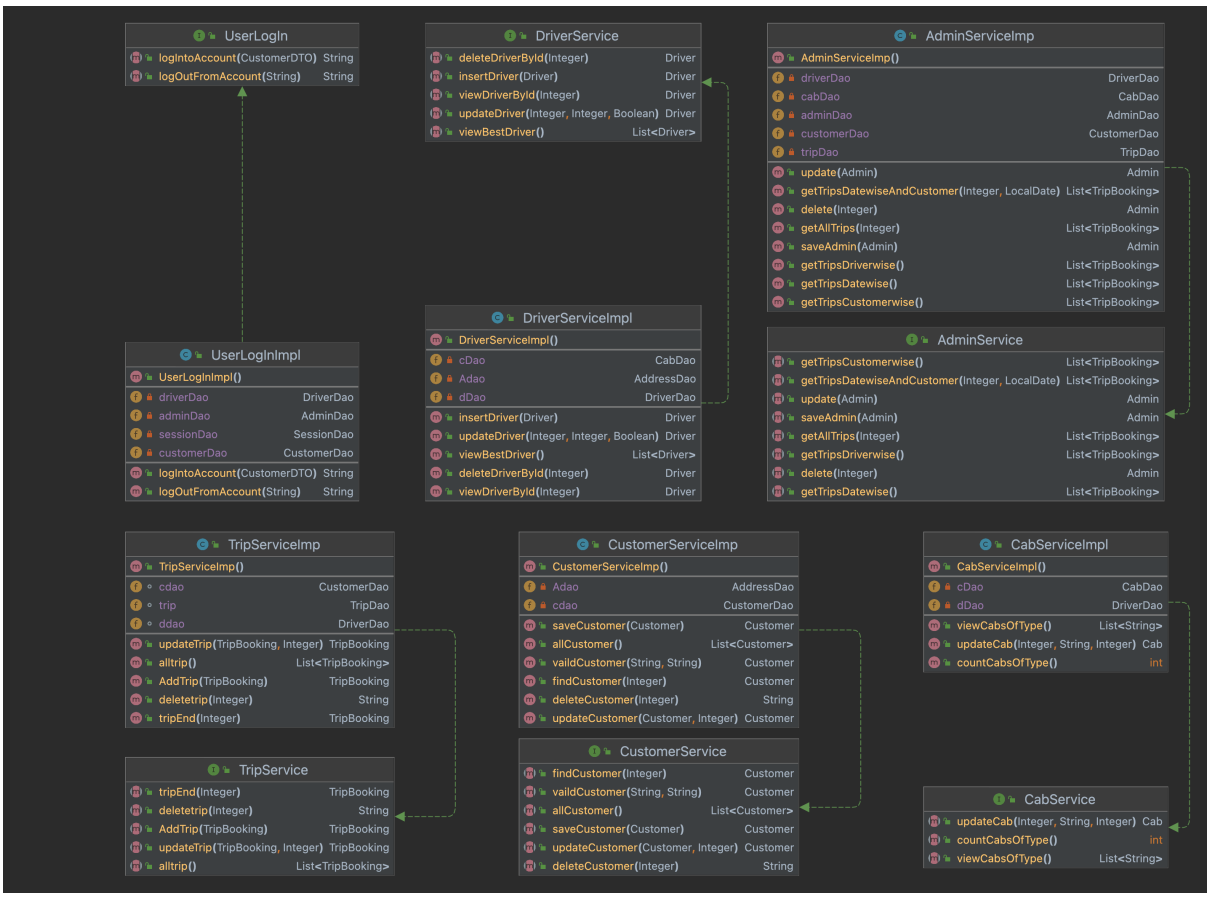

Figure 5.5.: Diagram of the service layer

5.2. Baseline

As we need some way to verify our results we decided to use also chose a baseline system. For this we decided to go with the euRent project which can be obtained by contacting the people at NSX at the contact information on https://normalizedsystems.org.

This project is of sufficient size and contains sufficient functionalities to represent an industrial level system. For completeness we provide a small overview of the technical details of this system in Table 5.2.

5.3. The transformation process

In what follows we will describe the general flow of transforming a plain Java system so it complies to the NST principles using μRadiant. We will end this section by providing a general overview of the μRadiant life cycle and provide some pointers on how to customize the generated code.

Files	47
Classes	46
Functions	118
Statements	220
Lines of code	1304
Comment lines	31 (2.3%)
Coverage	0%
Unit tests	1
Last commit	Aug 30, 2022

Table 5.1.: Technical specs of the cab booking application

Files	2523
Classes	2511
Functions	27853
Statements	73141
Lines of code	217788
Comment lines	82700 (27.5%)
Coverage	0%
Unit tests	92

Table 5.2.: Technical specs of the euRent project

5.3.1. Converting plain Java into NST

In order to start our transformation process, we first need to collect a list of all data elements we will need in our application. As our goal is to recreate the system as closely as possible we decided to only take into consideration the data entities which were already present without adding new ones unless absolutely necessary. To this end we primarily based ourselves on the Entity layer as described in a previous section and landed on following set of data entities

- Address: represents a physical address consisting off a street, number, zip code and city.
- Cab: the vehicles which can be booked by a customer. Each cab has a rate per km, a driver and a car type. In order to stay in line with the NST principles the car type has been placed in its own data entity.
- Car type: name of the type of car.
- Customer: this entity consists of a boolean flag to indicate whether the customer is on a journey and a link to the generic person object.
- Driver: a driver has a name, license number and rating. This object also contains a flag to indicate whether the driver is available or not, the cab he occupies and a link to the current trip if applicable. Just like the customer it has a link to a person entity.
- Payment: a payment currently only contains a flag to indicate whether or not it has been paid and the total amount which is due.
- Person: global person entity to identify users. It contains a username, password, email, mobile phone number and a link to a physical address. As user authentication and registration will in fact be handled by the account component which is provided in μRadiant, the username and password

fields could in fact be renamed and used to store other details such as a name and a date of birth. As this does not affect the core working however we decided to leave it as-is.

- TripBooking: a trip booking contains a customer who has booked a specific driver to transport him between two locations on two specific points in time. A trip has a certain amount of kilometers and will result in a payment which is linked.

These data entities were then created using μRadiant in a single component after which the first version of the code was generated.
By going through the other layers of the original system, ie controller, exception, repository and service, we noticed that apart from CRUD operations the only thing missing in the NST application would be a way to find specific data entities. Thankfully this can easily be resolved by adding finders to the data elements.
In the following subsections we will briefly go over how the μRadiant workflow works, how custom code can be injected between anchors in the generated code and how we can use finders and data projections to filter data objects.

5.3.2. On the μRadiant Flow

It is worth spending some time on how the cycle of generating an application with μRadiant works. In this cycle can distinguish six phases.

5.3.2.1. Model/Coding phase

As we start a new application we first need to model our application just as we did in the section above. Or, if an application has already been generated, we have been happily coding some custom extras. Once we are happy with the model we created/altered or think our code is ready for testing, we can proceed to the next phase.
A small remark: it is not possible to harvest changes after adding models. For this reason, good practice would be to harvest changes made to the code before making changes with the model editor.

5.3.2.2. Validation

After each remodel μRadiant will automatically validate the model. If any problems arise they can still be fixed and generation of non-working code is avoided.

5.3.2.3. Harvest

If any custom code was written which needs to be kept, don't forget to Harvest it. Harvesting will extract the custom code from between the relevant anchors and store it in the harvest folder located under the components directory. Once we had a successful harvest we can continue to the next phase.

5.3.2.4. Expand

The expansion phase is where the application is generated. During this phase two things happen. Firstly, all the existing code is deleted. Secondly, based on the defined models, settings and saved custom code,

a new application is generated. For coding purposes it is advised to open the pom.xml file from the /expansions/¡application¿ folder as a project in an IDE such as Intelij in stead of just opening the root folder as a project.

5.3.2.5. Build

The previous phases can already be executed in an endless cycle, for example if you forgot something or suddenly a more elegant solution came to mind. If however the code seems mature enough to be run, all we need to do before we can deploy is to build our application. This will trigger all necessary commands and scripts to produce a deployable application.

5.3.2.6. Deploy

Deploying an application requires Docker to start and maintain containers. Once the application has been successfully deployed it is time to start experimenting with the newly added functionalities and layouts. Once the application has been deployed we will inevitably end up in the first phase again and the cycle will start again.

5.3.3. On Anchors and Custom Code

In the code generated by the expanders a lot of anchors are present. These anchors denote places where things happen: where imports are located (on top of the file), where variables are located, the phases in which a method is divided, ...

Most of these anchors are there to provide clarity and while you can alter the code between them, any changes will be removed when the code is regenerated to ensure the four design principles are not violated. But, as it would be strange to not be able to write custom code at all, some anchors clearly indicate that between them you can safely write custom instructions. These anchors are denoted with "//anchor:custom-¡something¿:[start—end]".

Any custom code written between these anchors is extracted and saved to a safe place during the harvest phase.

5.3.4. On Finders, Projections and Custom Finders

As no application would be complete without some finding functionality, μRadiant is able to provide most of these by default. A expansive list of operators is provided with which you can filter out certain field-values. It is even possible to combine multiple operators and field-values to obtain more complex finders. As can be seen in Figure5.7 the name is automatically generated by adding field-operator pairs.

If however no combination of operators would satisfy the requirements, there is always the option of selection the 'isCustom' option. Doing so will add some small changes to the code which allow for the implementation of this custom finder. Implementing a custom finder happens according to following steps

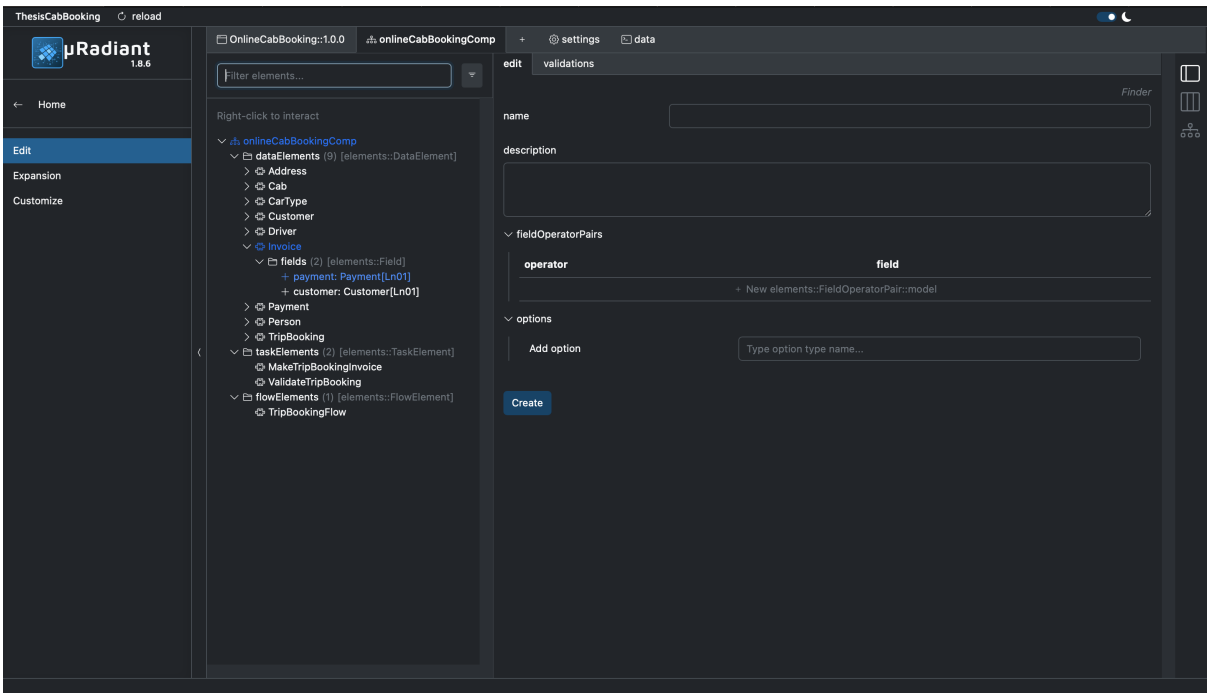

Figure 5.6.: Blank page for adding a new finder to a data element

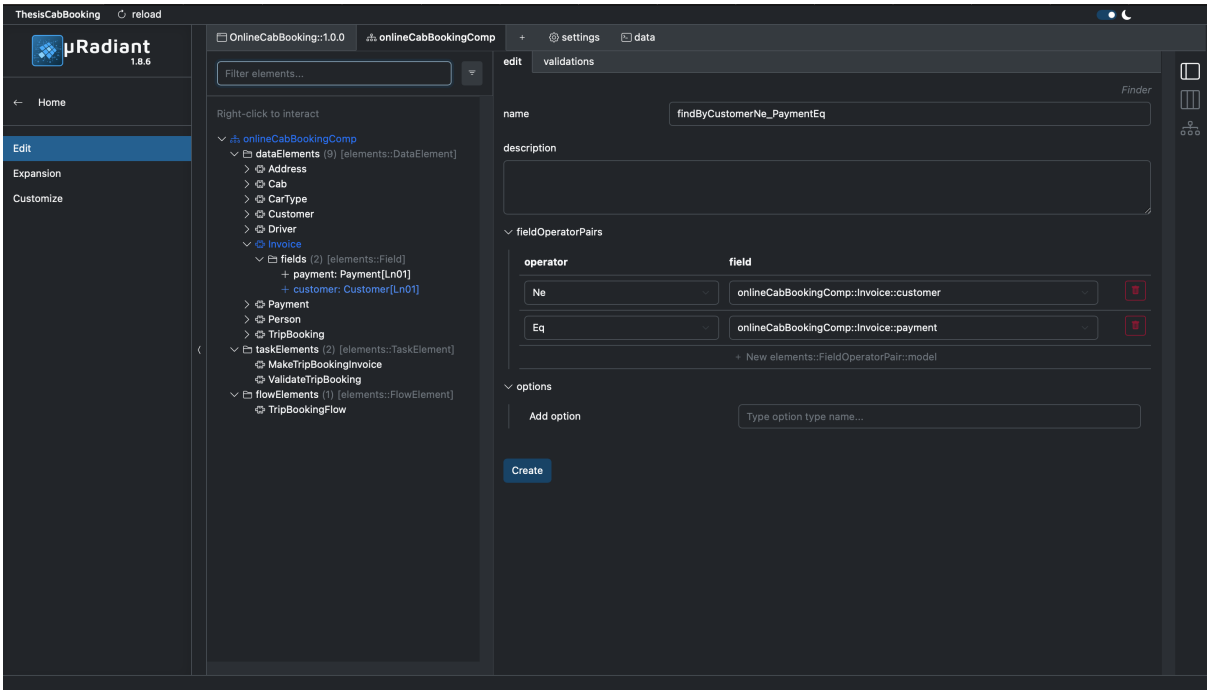

Figure 5.7.: Example finder for the invoice data element

- Add the finder in μRadiant and declare it as custom finder, otherwise the finder is put among the regular findBy-methods anchors and cannot be changed without manual overriding in the custom methods anchor, which would lead to code duplication.

- Add the necessary fields for your finder to the corresponding ¡finder¿Details class along with any additional getters and setters.

- Modify the mapping functions jsToViewModel and ViewModelTojs found in ¡finder¿-ko-mapper.js in the view layer.
- Create an appropriate form ¡finder¿-ko-form.html in the view layer.
- Add named queries to the ¡DataElement¿Data class in the data layer.
- Use these named queries in the ¡DataElement¿FinderBean to fetch the data for the finder.

A final item that needs some attention is the presence of data projections. DataProjections define a set of fields from their parent DataElement. When data is passed through the application, it is always represented in the form of a projection. There are 3 projections that are always present for each DataElement.

- Details Projection: The details projection serves as the default representation for most backend logic and is used to create and modify. The details projection is defined by the set of all fields in the DataElement excluding the calculated Fields.
- Info Projection: The info projection is a reduced set of fields from the DataElement. It defines the columns in the table in the user interface by showing the fields which have been marked as infoField.
- DataRef projection: The DataRef projection is used in several places to reference an instance of a DataElement without requiring the entire DataElement projection. By default, this projection only contains the id (and name if possible) of that instance. It is also possible to define your own Fields to add to the DataRef To do so, add the option functionalKey with the names of the fields separated with _. This functionalKey option generates a DataRef class which contains said Fields in addition to the name and id, so they can be used to look up instances. This new dataRef class will then be used when fetching data from the dataBase. Adding this option forces all functional fields to be contained in each projection.

Developers can also define their own projections. These projections define a set of reference fields, which point to fields of the DataElement.

5.4. Evaluating quality properties

As mentioned previously we will evaluate our systems based on metrics obtained by running SonarQube on our codebase. In what follows we will briefly go over how we will evaluate the key quality properties we defined in section 3.2.1, section 3.2.2 and section 3.2.3.

5.4.1. Stability

For this thesis we decided to measure stability of a system based on the number of bugs and vulnerabilities. In concrete we will use the reliability and security metrics which are provided by SonarQube.
We decided to use these metrics specifically because of the way in which they are related to stability. For instance, if a software system contains a large amount of bugs with respect to the total amount of code it is not to be considered reliable and, as buggy code will affect the way in which the software functions, it will also impact the stability. A similar argument can be made for security metrics: a system which contains security issues is vulnerable to cyber attacks, leading to decreased performance and instabilities.

5.4.2. Maintainability

In order to measure the maintainability of our systems we will take a look at the amount of code smells, derived from that the total amount of technical debt and the cognitive complexity.
As code smells provide an indication of programming mistakes design-wise, they also tell us where future problems or bugs may arise so we can fix them proactively. The total amount of estimated effort to fix all these smells is what we call the technical debt of the software: codesmells can be seen as small loans on the maintainability of a software and at some point in the future we will have to repay this debt. The cognitive complexity of a system is an indication of how difficult a system is to understand and software that is hard to understand is more difficult to maintain.

5.4.3. Extensibility

In this thesis we will measure the extensibility of a system by how easy it is to add new features or extend existing features. For instance, how many files do we need to edit in order to implement certain functionalities, can we automate certain update procedures, how easy is it to change a used technology, ...?

5.5. Future work

- Testing: at time of writing there is no functionality to automatically generate tests. As unit tests can be generated automatically using an IDE like InteliJ or plugins like Devmate, it might be a useful extension in future development of the expanders. These generated tests can then be placed in a testing suite so they can later be used as regression tests when new functionality is added.
- Automatic import: currently there is only functionality to import existing NS projects. It might be useful to provide functionality which allows for automatic importing of data elements as most software architectures already provide some separation of data entities, business logic and visual representation.

6. Comparison

In what follows we will compare the results obtained by running SonarQube on both the original system and the NST translation. We will do so by discussing each metric separately and explain the results. To provide some baseline as to what metrics are to be expected from a NST system that has been through active development we also include metrics obtained on the euRent project as provided by NSX.

Metric	Original system	NST system	NST example system
Bugs	3	0	16
Reliability rating	C	A	C
Reliability remediation effort	35m	0m	4h15
Cognitive Complexity	26	2104	19724
Code smells	120	3124	25187
Technical debt	6h26	35d	263d
Debt ratio	1.0%	2.2%	1.9%
Maintainability rating	A	A	A
Vulnerabilities	8	0	0
Security rating	D	A	A
Security remediation effort	1h20	0	0

Table 6.1.: Metrics obtained by SonarQube

6.1. Reliability

Metric	Original system	NST system	NST example system
Bugs	3	0	16
Reliability rating	C	A	C
Reliability remediation effort	35m	0m	4h15

6.1.1. Bugs

According to the SonarQube scan, the original version of the project contains 3 bugs which they classify as major. These bugs are

- In exception/GlobalExceptionHandler, the getFieldError() method could throw a NullPointerExcpetion and currently this behavior is not anticipated. Remedying this bug is estimated to take about 10 minutes.

- In service/UserLogInImpl 2 bugs are present, both in the logIntoAccount method. The first one refers to an issue on line 46 where the ID is being retrieved from an object obtained by using the get() method from the base Java library. As this method may throw a NoSuchElementException and this behavior is not anticipated, this is considered a major issue.
  The second bug refers to the if-statement on line 50. This statement tries to resolve the before mentioned issue, but in the current implementation it makes little sense to have the check at that location. According to SonarQube remedying these bugs will take 10 and 15 minutes respectively.
  A possible solution to resolve both bugs would be to refactor the method as follows

```
public String logIntoAccount(CustomerDTO userDto) {
    Optional<Customer> opt_customer = customerDao.findById(userDto.getUserId()
    );
    //  Optional<Driver> opt_driver = driverDao.findById(userDto.getUserId());
    //  Optional<Admin> opt_admin = adminDao.findById(userDto.getUserId());
    Integer userId = opt_customer.get().getUserId();
    Optional<CurrentUserSession> currentUserOptional = sessionDao.findById(
    userId);
    if (!opt_customer.isPresent()) {
      throw new AdminExceptions("user not found");
    }
    if (currentUserOptional.isPresent()) {
      throw new UserAlreadyExistWithuserId("User already logged in with this
    number");
    }
    if (opt_customer.get().getPassword().equals(userDto.getPassword())) {
      String key = RandomString.make(6);
      CurrentUserSession currentUserSession = new CurrentUserSession(
    opt_customer.get().getUserId(), key,
          LocalDateTime.now());
      sessionDao.save(currentUserSession);

      return currentUserSession.toString();
    } else {
      throw new InvalidPasswordException("Please Enter Valid Password");
    }
}
```

```
public String logIntoAccount(CustomerDTO userDto) {
    Optional<Customer> opt_customer = customerDao.findById(userDto.getUserId()
    );
//    Optional<Driver> opt_driver = driverDao.findById(userDto.getUserId());
//    Optional<Admin> opt_admin = adminDao.findById(userDto.getUserId());
    if (opt_customer.isPresent()) {
      Integer userId = opt_customer.get().getUserId();
      Optional<CurrentUserSession> currentUserOptional = sessionDao.findById(
    userId);
      if (currentUserOptional.isPresent()) {
        throw new UserAlreadyExistWithuserId("User already logged in with this
     number");
      }
      if (opt_customer.get().getPassword().equals(userDto.getPassword())) {
        String key = RandomString.make(6);
        CurrentUserSession currentUserSession = new CurrentUserSession(
    opt_customer.get().getUserId(), key,
            LocalDateTime.now());
        sessionDao.save(currentUserSession);

        return currentUserSession.toString();
      } else {
        throw new InvalidPasswordException("Please Enter Valid Password");
      }
    }
    else {
      throw new AdminExceptions("user not found");
    }
}
```

The NST version of the same project currently does not contain any bugs. To verify whether or not this is inherent to NST we verify this result with the euRent system. This system was found to contain 16 bugs and are distributed among the different components as follows

- account: Change the 'if (deletePermanent)' condition so that it does not always evaluate to "false" in following places
  - DataAccessCruds.java in delete method on line 448
  - UserCruds.java in delete method on line 469
  - UserGroupCruds.java in delete method on line 422

  As this code is generated by the expanders - it is located between regular start-end anchors - this needs to be resolved during development of the expanders.

- rentwork: 0 bugs found.
  As this is the real 'custom' component in the application this is the number is the most important: finding no bugs in a component in which developers have actively added custom code is an indication that NST provides a framework which is robust enough to withstand manual adapting.

- utils: 3 bugs found, all in ExecutorImpl.java.
  - Refactor ^\\w+(\\\\\\w+)*(\\.\\w+)?$ that can lead to a stack overflow for large inputs on line 46. This regular expression is intended to verify that the argument refers to a valid filename, so in practice this problem is very unlikely to arise. However, for safety measures we can limit the maximum allowed size by adding '1,255' to our regex.
  - Refactor \^\\w+(\\.\\w+)*\$ that can lead to a stack overflow for large inputs on 58. The same reasoning as above holds here as well.
    The checking on regexes is custom code and thus needs to be fixed by a developer.
  - Either re-interrupt this method or rethrow the "InterruptedException" that can be caught at line 69. What this tells us is that we need to provide more specific catch clauses in stead of one generic catch all. As this code is generated by the expanders - it is located between regular start-end anchors - this needs to be resolved during development of the expanders.

- validation: 6 bugs were found, all of them being 'This class overrides "equals" and should therefore also override "hashCode"'. To resolve these bugs, the hashCode method should be implemented in Time.java, KBO.java, Iban.java, Email.java, DateTime.java and DateTime.java. The most elegant way of resolving these bugs is to alter the expanders to generate the hashCode method whenever the equals method is overridden.

- workflow: in this component 4 bugs were found. The 'Either re-interrupt this method or rethrow the "InterruptedException"' bug was found 3 times: once at line 231 in ExecutorServiceTaskProcessor.java and twice in ParallelTaskProcessor.java at line 117 and 249. The final bug on line 20 in StateTaskParameterContextFactory.java refers once again to the limiting of size when using a regex as seen in the utils component. As the concerned files are located in the ext folder it means this is full custom code and resolving the bugs is up to a developer.

Conclusion

Of the 16 bugs found in the baseline system, 10 need to be resolved by altering the expanders. The remaining 6 bugs are due to developers and are divided equally between either providing more specific catch clauses or limiting the allowed size of a regex. The original Java project contained only 3 bugs,

but when we look at the problems that might arise when the application is actually running it is safe to conclude that those found in the Java project are potentially more harmful than those in the NST example project. Our NST conversion was found to be bug free.

6.1.2. Reliability rating and remediation

Due to the bugs mentioned in the previous point, the original Java project has received a reliability rating of C. The total time to resolve all bugs and go from C to A is estimated to be 35 minutes, which is still an acceptable amount of time given the nature of the bugs.
The NST version of the project already has a rating of A as there are no bugs found.

Conclusion

When it comes down to bugs, using NST limits the possibility for introducing them due to the use of expanders during code generation/rejuvenation.

6.2. Cognitive complexity

Metric	Original system	NST system	NST example system
Cognitive Complexity	26	2104	19724

The cognitive complexity of an application refers to how difficult it is to understand it. According to the white paper which can be retrieved at https://www.sonarsource.com/resources/cognitive-complexity/, the cognitive complexity is increased when encountering loop structures, conditionals, catches, nested structures, switches and recursion. For example, the use of a switch statement with multiple cases will result in a lower cognitive complexity then when using a if-else if structure.

The original system has a cognitive complexity of 26, all of which is located in the service layer of the application, while the NST version has a cognitive complexity of 2104 in the custom module with following distribution

Control	468	Data	856
Logic	152	Proxy	416
Shared	212	View	0

The other components such as account, workflow etc are not explicitly mentioned by SonarQube, nor are they present in the codebase.
As for the example NST project we can create following matrix

Component	Control	Data	Logic	Proxy	Shared	View	total
account	866	1370	497	692	511	-	3936
rentWork	2344	4198	1129	2116	952	-	10739
utils	302	420	158	308	152	-	1340
validation	110	87	19	52	179	-	447
workflow	591	976	740	572	383	-	3262

Converting these numbers to percentages gives us

Component	Control%	Data%	Logic%	Proxy%	Shared%	View%	total%
account	22	34.8	12.63	17.58	12.99	-	100
rentWork	21.83	39.1	10.51	19.7	8.86	-	100
utils	22.54	31.34	11.79	22.98	11.34	-	100
validation	24.6	19.46	4.25	11.63	40.04	-	100
workflow	18.12	29.92	22.68	17.53	11.74	-	100

When we look at the larger example NST project, we see that all components except validation follow a similar distribution pattern as our smaller NST project: the bulk of the cognitive complexity is located in the Data layer followed by Control, Proxy, Shared and Logic.
In general, this high complexity is due to the way in which code is currently generated by the expanders. For example, in the create method from the CabCruds class try-catch structures are used in combination with nested conditionals. This use of if-else structures can be justified by the need for anchor points between which developers can add custom code. Listing6.2 shows the method mentioned above.

```
1  // anchor:create:start
2  public CrudsResult<DataRef> create(ParameterContext<CabDetails> detailsParameter)
     {
3    if (sessionContext.getRollbackOnly()) {
4      return getDiagnosticHelper().createCrudsError(CREATE_ERROR_MSG_KEY);
5    }
6    Context context = detailsParameter.getContext();
7    CabDetails details = detailsParameter.getValue();
8    // @anchor:preCreate:start
9    // @anchor:preCreate:end
10   // anchor:custom-preCreate:start
11   // anchor:custom-preCreate:end
12   // @anchor:preCreate-validation:start
13   // @anchor:preCreate-validation:end
14   CabData cabData = new CabData();
15   try {
16     // @anchor:create-beforeProjection:start
17     // @anchor:create-beforeProjection:end
18     // anchor:custom-create-beforeProjection:start
19     // anchor:custom-create-beforeProjection:end
20     detailsProjector.toData(cabData, detailsParameter);
21     // @anchor:create-afterProjection:start
22     // @anchor:create-afterProjection:end
23     // anchor:custom-create-afterProjection:start
24     // anchor:custom-create-afterProjection:end
25   } catch (Exception e) {
26     sessionContext.setRollbackOnly();
27     if (logger.isErrorEnabled()) {
28       logger.error(
29           "Cannot fill data object", e
30       );
31     }
32     return getDiagnosticHelper().createCrudsError(CREATE_ERROR_MSG_KEY);
33   }
34   // @anchor:implicitNameFieldOnly-beforePersist:start
35   String identity = details.getName();
36   if (identity != null) {
37     identity = identity.trim();
38     identity = identity.length() > 0 ? identity : null;
39   }
40   // @anchor:implicitNameFieldOnly-beforePersist:end
```

```
41      try {
42        entityManager.persist(cabData);
43        // @anchor:implicitNameFieldOnly-afterPersist:start
44        if (identity == null) {
45          identity = "C-" + cabData.getId().toString();
46        }
47        cabData.setName(identity);
48        entityManager.flush();
49        // @anchor:implicitNameFieldOnly-afterPersist:end
50        if (cabData != null) {
51          if (logger.isDebugEnabled()) {
52            logger.debug(
53                "Created CabCruds with { id : " + cabData.getId() + ", name: " +
    cabData.getName() + " }"
54            );
55          }
56        }
57      } catch (Exception e) {
58        sessionContext.setRollbackOnly();
59        if (logger.isErrorEnabled()) {
60          logger.error(
61              "Cannot perform entry creation", e
62          );
63        }
64        return getDiagnosticHelper().createCrudsError(CREATE_ERROR_MSG_KEY);
65      }
66      CrudsResult<DataRef> result = getDataRefFromData(detailsParameter.construct(
    cabData));
67      // @anchor:postCreate:start
68      // @anchor:postCreate:end
69      // anchor:custom-postCreate:start
70      // anchor:custom-postCreate:end
71      return result;
72    }
73    // anchor:create:end
```

Listing 6.1: Create method with high congitive complexity

Conclusion

When it comes to cognitive complexity the current way of code generation in NST is not desirable and based on this metric an NST application should not be used in an industrial setting.

6.3. Maintainability

Metric	Original system	NST system	NST example system
Code smells	120	3124	25187
Technical debt	6h26	35d	263d
Debt ratio	1.0%	2.2%	1.9%
Maintainability rating	A	A	A

6.3.1. Code smells

Our scan of the original project revealed the presence of 120 code smells in the code base. Some of the revealed tags are

Code smell	Original system	NST version	NST example
Unused: these code smells refer to deletion of commented code blocks and unused imports.	38	2k	17k
Clumsy: clumsy code smells indicate situations that could have been implemented in more efficient ways, such as specifying types in constructors via diamond constructors, immediately returning expressions in stead of storing them in temporary variables and using appropriate methods to check for emptiness.	38	146	1.4k
Convention: reports violations of coding conventions such as formatting and naming.	29	283	227
Bad practice: these issues will work as intended, but the way in which they are implemented are acknowledged to be bad practice.	3	101	858
CERT: reports violations of CERT standard rules as can be found at https://wiki.sei.cmu.edu/.	5	226	2.2k
CWE: relates to rules in the Common Weakness Enumeration	16	258	2.4k
Design: reveals questionable design choices, such as having duplicates of literals in stead of using a variable for it.	4	80	632

Fixing all code smells in the original system would take an estimated 6h 26 minutes while the NST reproduction would require 35 days of work. The example NST system would require a staggering 263 days to resolve all code smells.

It is actually quite surprising that the systems following the NST principles have such a high number of code smells, so let's take a closer look at some of the categories.

- Convention: almost all issues with this tag in the NST reproduction refer to the renaming of the package from cabBookingCore to cabbookingcore so it complies with naming conventions. This can easily be done in the μRadiant editor and can be fixed in less than 5 minutes. The estimated time to fix all these issues is estimated to be 5 days and 5 hours.
  In the example NST system almost all issues with this tag refer to issues in which a field or method needs to be renamed so it corresponds to Java naming conventions.

- Unused: SonarQube reports over 2k code smells of unused code. The main issue here is that, while these are indeed correctly identified as currently unused, not all of them can be removed due to the way in which the code is generated by the expanders. For example, imports such as shown in Figure6.1 and Figure6.2 are included on a "these are most commonly used in this context" basis. Therefor it's not possible to resolve these smells as they are created by design. This means that of the estimated 35 days needed to resolve the code smells, 12 will always be there where the NST reproduction is concerned. For the example NST system these values can be scaled to 37 days of permanent outstanding remediation cost of a 108 total cost.

- CWE: the majority of these issues fall in one of three categories. A first category is one where the different agents created to provide access to the different objects should be declared as either transient or serializable. A next category would be one removal of deprecated methods. A final category is avoiding the use of generic exceptions such as RuntimeException.
  Code smells that fall in either of the first two categories can be resolved by updating the expanders. Code smells that fall in the third category are harder to resolve as they would require the automatic creation of per-class exceptions which can then be further customized by the developer. This would lead to an exponential increase in number of classes which is both impractical and undesirable.
  In total it would take 9 days and 7 hours to resolve all smells with the CWE tag on the NST reproduction. If we take into account that about a third of the workload cannot be resolved by design, there will always be an outstanding cost of at least 3 days.

- Bad practice: all code smells with this tag can be resolved by either adding a @deprecated tag or @Override annotation in the expanders. SonarQube gives an estimated cost of 1 day on the NST reproduction and 9 days on the example NST system.

- Design: all except one of the design code smells refer to the use of duplicate string literals. The estimated cost for fixing these smells is 1 day 6 hours on the NST reproduction and 16 days on the example NST system.

6.3.2. Technical debt and Maintainability Rating

As stated in the previous paragraph, it would take 35 days to fix all code smells from our NST system. We can solve almost 6 days of work in less than 5 minutes using μRadiant and an additional 9 days of work can be taken care of by tweaking the expanders. Of the 35 days needed to fix all code smells at least 15 days can not be resolved due to the design of NST.

In the previous discussion we have not mentioned CERT and clumsy code smells. The reason for this is that CERT smells are almost always tagged together with CWE and their remediation cost has thus already been counted. The clumsy smells require a case-by-case inspection: some could be fixed quite easily in the expanders, such as merging nested conditional statements, while others cannot be resolved due to the presence of anchors as is the case in for instance the getCustomDisplayName method from the CarTypeCruds class. As these smells only account for a workload of 7 hours to remedy and overall they are classified as minor issues, it might not be feasible to spend time at tweaking the expanders just for the sake of reducing code smells. The remaining 5 days of work are distributed among various other tags which we will not discuss here for sake of interest.

Our initial system on the other hand can be released of its code smells in less than 7 hours.

The maintainability rating is closely related to the technical debt ratio. As both versions have a debt ratio below 5%, all systems get a maintainability rating of A.

Conclusion

While systems generated in compliance with the NST principles at first glance stink to high heaven, when we take a closer look not everything is as bad as it seems: most of the code smells can be resolved by tweaking the expanders. We know that when left unsupervised projects will accumulate technical debt over time.[tecb][teca] The main benefit of NST is that, since the majority of the code is generated/rejuvenated by μRadiant, the technical debt will almost never increase as fast as with pure manual system development.

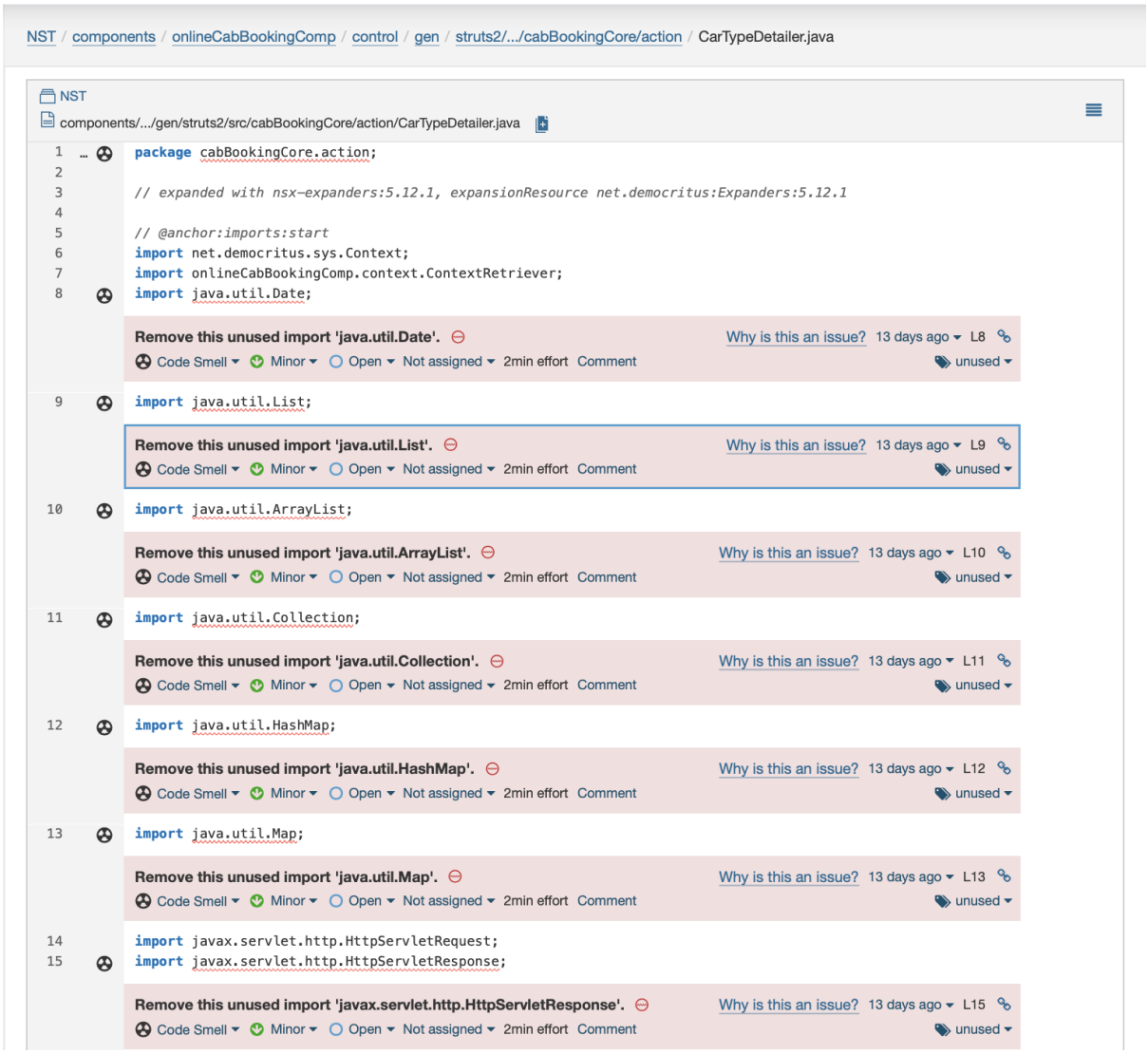

Figure 6.1.: Example of the unused imports generated by μRadiant

Based on this metric and the discussion above, using NST is preferable over manual development.

6.4. Security

Metric	Original system	NST system	baseline system
Vulnerabilities	8	0	0
Security rating	D	A	A
Security remediation effort	1h20	0	0

Figure 6.2.: Example of the unused imports generated by μRadiant

6.4.1. Vulnerabilities

In our initial system, SonarQube found 8 vulnerability issues, all of which entail replacing a persistent entity with a POJO or DTO object. Our NST systems come with zero vulnerability issues.

6.4.2. Security rating and remediation

As all of the vulnerabilities found in the initial system are deemed to be critical issues, this version has received a security rating of D which is the second last lowest rating. Luckily the estimated remediation effort is 1 hour 20 minutes, so we can easily improve this rating from D to A.

Since our NST version has no detected vulnerabilities, it received a rating of A.

Conclusion

As NST generates code using technologies which are kept up-to-date, it is nearly guaranteed that security issues will never arise. It is clear that based on this metric using NST is preferable.

6.5. Comparison using CodeScene

6.5.1. Comparing based on CodeScene

As it might prove useful to look at the system versions from multiple angles we will take a look at how each versions measures according to CodeScene.

When run on both the old and NST version, on first glance there didn't seem to be any significant results. For example, the code health shown in Figure6.3 reveals no issues with code health while the NST version is deemed to have some issues as shown in Figure6.4.

If we dive deeper into the different hotspots of our NST application, it turns out the the three hotspots

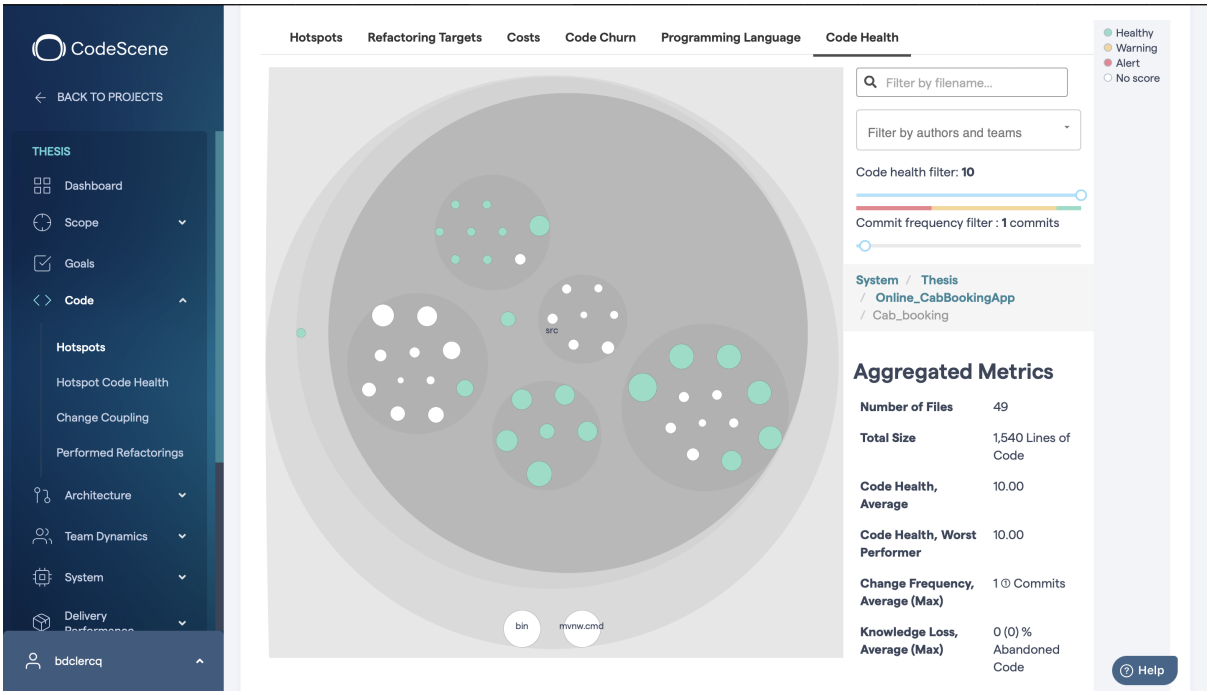

Figure 6.3.: Code health of original system according to CodeScene

which contain red zones are actually part of JavaScript libraries and therefore fall outside the scope of NST. When we shift our focus to the part we generate using μRadiant we see that, while most of the code seems to be healthy, we do have some slight issues as revealed in Figure4.1. These issues in components/onlineCabBookingComp/ are as follows

- data layer: TripBookingCruds.java
  - Low cohesion in the module as measured with the LCOM4 metric (Lack of Cohesion Measure). With LCOM4, the functions inside a module are related if a) they access the same data

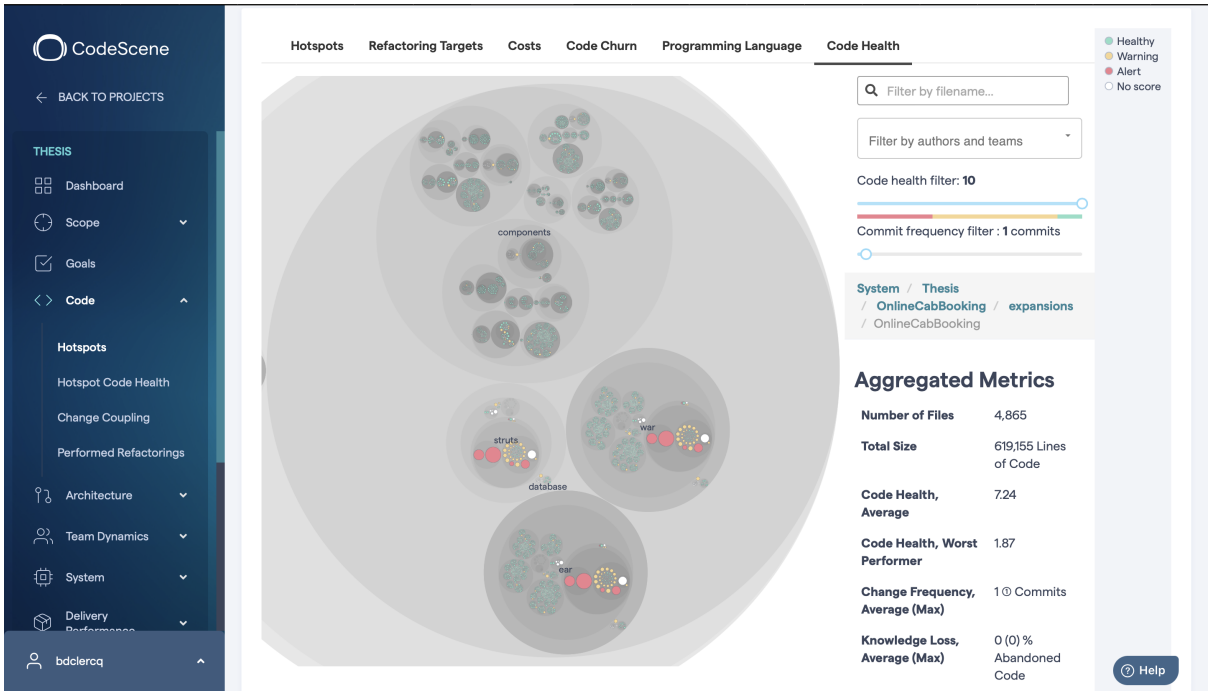

Figure 6.4.: Code health of NST system according to CodeScene

members, or b) they call each other. High Cohesion is desirable as it means the Separation of Concerns theorem is followed as it should be. CodeScene has found that this module has at least 4 different responsibilities amongst its 47 functions.
If we take a look at the X-ray of the module however, CodeScene tells us “There’s no significant structural recommendations to make. Keep up the good work!”.

  - The create method has a cyclomatic complexity of 11, which is deemed to high.

  - Duplicate code blocks have been detected. Seeing as these code blocks are generated by the expanders, without requiring any work from the developer, these duplicate code block issues can be ignored.

- shared layer

  - ComponentMetaData.java

    * Low cohesion in the module as measured with the LCOM4 metric (Lack of Cohesion Measure). With LCOM4, the functions inside a module are related if a) they access the same data members, or b) they call each other. High Cohesion is desirable as it means the Separation of Concerns theorem is followed as it should be. CodeScene has found that this module has at least 4 different responsibilities amongst its 13 functions.

    * Once again duplicate code blocks have been detected.

    * There are 4 methods which are too large and should be split up, these methods are getDataElementDef_TripBooking(LoC = 89 lines), getDataElementDef_Driver(LoC = 75 lines), getDataElementDef_Person(LoC = 72 lines), getDataElementDef_Address(LoC = 70 lines).

  - TripBookingMapper.java

* This file has 2 bumpy roads, which are functions that contain multiple chunks of nested conditional logic inside the same body. The deeper the nesting and the more bumps, the lower the code health. CodeScene considers the following rules for the code health impact: 1) The deeper the nested conditional logic of each bump, the higher the tax on our working memory. 2) The more bumps inside a function, the more expensive it is to refactor as each bump represents a missing abstraction. 3) The larger each bump – that is, the more lines of code it spans – the harder it is to build up a mental model of the function. The nesting depth for what is considered a bump is 2 levels of conditionals. The reported bumpy roads are convertToMap and convertToDetails.

* Complex methods: the same methods reported in the item above have a cyclomatic complexity of 19 and 20 respectively and should be simplified. As in other components these methods are also reported to have a too high cyclomatic complexity, it might be something to consider changing in the expanders.

* The convertToDetails method is reported to be too long and should be split up into smaller routines. As this is also a recurring remark in other components it might be beneficial to try and tackle this issue as well.

- view layer: tripBooking-finders.js, person-finders.js, driver-finders.js. In all these files the different finders which are generated are reported as being duplicate code blocks.

6.5.2. Analyzing the baseline

If we look at Figure 6.5 we see how our baseline performs. On average the code has a health rating of 8.59 which is very good, and from observing the health distribution it becomes clear that the problems are situated in one specific component of the application. When we zoom in on this component it turns out it is the one containing the JavaScript lib files. **Conclusion**

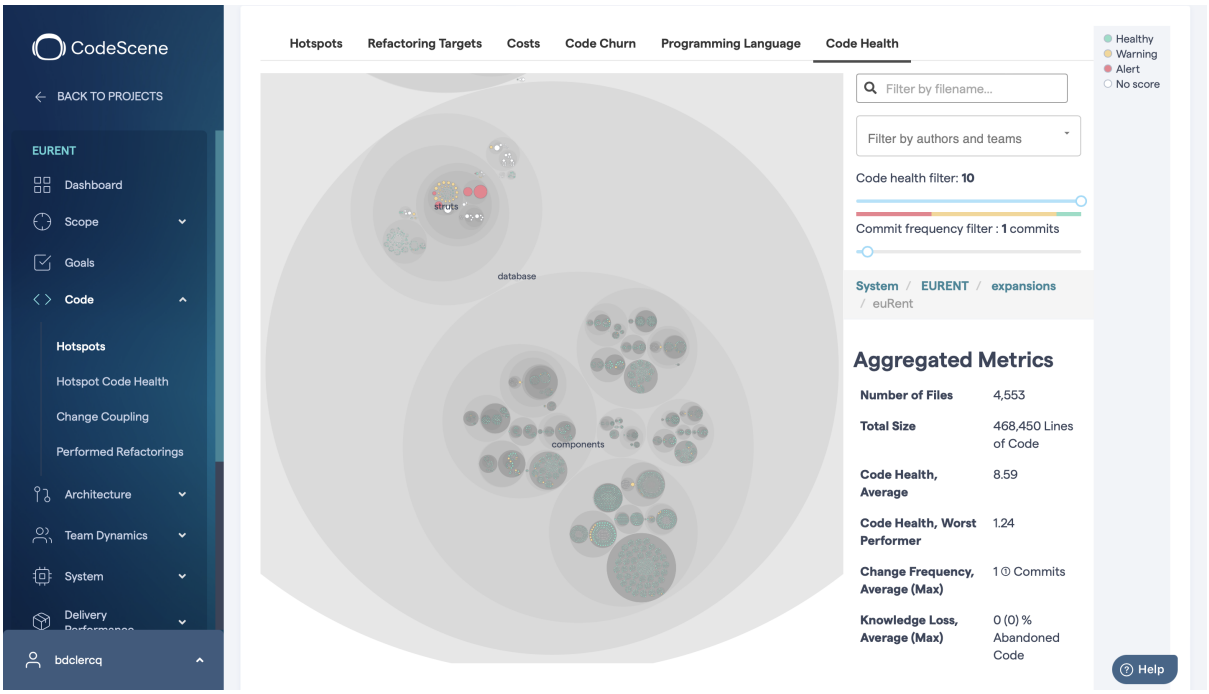

Figure 6.5.: Code health of the baseline system according to CodeScene

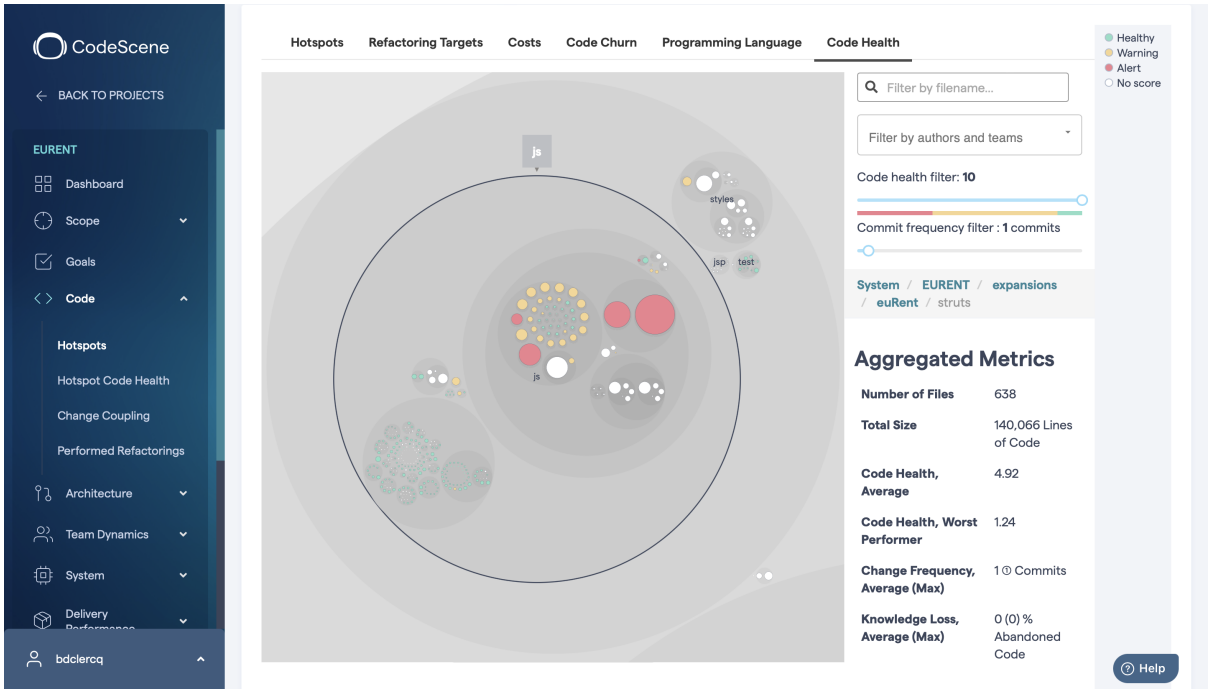

Figure 6.6.: Code health of the JS lib component

Based on the analysis by CodeScene, all systems are more or less equal. We see that our NST systems do contain some hotspots with very poor health, but on closer inspection it turns out they belong to an external library and thus fall outside of the scope of this thesis as they don't intrinsically define NST.

6.6. Extensibility

When it comes to extensibility of our system we decided to take a look at something that was currently missing in our system under observation: the automatic generation of invoices.

In order to implement this functionality in our NST system we need to take following actions

- Create a data element and task element in μRadiant.
- Regenerate the code.
- In the running application, add the newly created task to the existing workflow.
- Create a custom implementation in the ¡task element¿Impl or ¡task element¿Impl2 file in the Logic Layer.

If we wanted to add the same functionality to our system under observation we would have to perform following steps

- In the entity layer we need to declare our data model.
- In the controller layer we need to implement the CRUD operations for our invoices.
- In the repository layer we need to define the API which the system can use to access our back-end.

- In the service layer we need to extend the AddTrip method to include the generation of invoices. Doing so will violate the Separation of Concerns principle and thus contribute to instability of the software, which means that in order to keep our system stable we would need to refactor our system.

Conclusion

It is clear that adding this seemingly small feature would require little work in the NST system, but would require a major overhaul in the existing system if we do not want to take a loan on the stability of our software.

6.7. Conclusion

Let us summarize the above intermediate conclusions

- Stability: systems generated by μRadiant are more reliable than traditional Java systems and the NST systems were found to contain no vulnerabilities.
- Complexity: our generated NST systems have an exponentially higher cognitive complexity, making them more difficult to read and understand.
- Maintainability: code generated by μRadiant contains a staggering amount of code smells, most of which can be resolved by tweaking the expanders, and due to the way in which μRadiant generates/rejuvenates code the code smells can easily be kept under control.
- Extensibility: it would require less effort to add new functionality in our NST system than in the system under observation.

It becomes clear that making the transition from old Java systems to NST based systems may come at an initial cost, but over time it will certainly be worth the investment and will lead to software that is indeed maintainable, stable and extensible.
A sidenote that needs to be made is the following: if there is no access to μRadiant NST just becomes another of the available architectures which require the necessary discipline to follow and not take shortcuts. This means that, if the organisation does not want to invest in licenses for μRadiant and the necessary training, it is not worth the time and effort switching to NST.

Metric	Manual development	NST code generation
Reliability	0	1
Cognitive Complexity	1	0
Maintainability	0	1
Security	0	1

Table 6.2.: Overview of intermediate conclusions per metric

While at first glance the NST version of our Java system looks to be way worse than our initial version - according to SonarQube we would need well over a month to remedy all issues! - things aren't as bad at all.

Indeed, if we take a closer look at the two areas in which NST seems to perform poorly - namely cognitive complexity and code smells - it becomes clear that many of the issues can be resolved by tweaking the expanders by for instance switching from try-catch statements to switch statements in or-

der to lower the cognitive complexity or resolve 8 days of remediation cost from code smells, thereby providing cleaner code.

The main question here would be the following: is it worth recreating a system using the μRadiant tool? The answer is: it depends.
If an architectural blueprint is present - which really should be the case in a professional setting - it should not take long to transform an application using μRadiant. The advantage obtained from doing so, i.e. obtaining a system build according to the four design principles given in section 3.2.1 which can be mathematically proven to deliver software with the desired properties, outweigh the initial high cognitive complexity and high amount of code smells.[VM03]

A. Bibliography

Bibliography

[Bab21] Shibu Babuchandran. Software release life cycle (srlc): Understand the 6 main stages, 2021. [Online; publication date Dec 6 2021]. 26

[DSN03] Serge Demeyer, Ducasse Stephane, and Oscar Nierstrasz. *Object-oriented reengineering patterns*. Morgan Kaufman Pub., 2003. 11, 19

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

[Ext21] Extensibility. Extensibility — Wikipedia, the free encyclopedia, 2021. [Online]. 26

[Fow19] Martin Fowler. *Refactoring Improving the Design of Existing Code*. Pearson, 2019. 19, 21

[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

[Mar18] Robert C. Martin. *Clean Architecture A CRAFTSMAN'S GUIDE TO SOFTWARE STRUCTURE AND DESIGN*. Pearson, 2018. 25

[Mod21] Model-View-Controller. Model-view-controller — Wikipedia, the free encyclopedia, 2021. [Online]. 8

[NGI] NGINX Glossary. What is load balancing? [Online]. 8

[Pie] Michael Pierce. Software release life cycle (srlc): Understand the 6 main stages. [Online; no publication date]. 25

[Ric15] Mark Richards. *Software Architecture Patterns Understanding Common Architecture Patterns and When to Use Them*. O'Reilly, 2015. 8

[sof] What is software rot, and why does it happen? [Online; no publication date or author]. 25

[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]. 25

[teca] What is technical debt? [Online; no publication date or author]. 49

[tecb] What is technical debt and what should you do about it? [Online; no publication date or author]. 25, 49

[VM03] Jan Verelst and Herwig and Mannaert. *Normalised Systems Theory*. Morgan Kaufman Pub., 2003. 13, 25, 57

B. Previous research projects

B.1. Research project 1

Title/subject: Sketch recognition
Promotor: Hans Vangheluwe
Related to this thesis: No.
Abstract: Sketch recognition is a problem that has been around for decades, and over the years several solutions have been proposed to resolve it. In this report we will first take a look at some of the earlier work that has been performed on this problem. The second part of will go into more detail about a prototype build based on one of these previous works, describing how we can implement it as well as providing some background information about the different aspects involved. To conclude we will go over the current limitations and provide possible improvements.

B.2. Research project 2

Title/subject: Visualizing large TCR networks
Promotor: Pieter Meysman and Sofie Giellis
Related to this thesis: No.
Abstract: In this research project, the aim was to find ways to efficiently visualise large TCR networks. To this end some of the available tools and frameworks were explored before selecting two viable options, Igraph and NetworkX. After some experimentation Igraph remained as the final option.