UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL DE CALCULATOARE

SASPS PROJECT

Advantages and disadvantages of using the service or library for DRY compliance

Mărăcine Mihail-Robert

2023

CUPRINS

1 INTRODUCERE

1.1 Context

In software development, one of the fundamental software principles is DRY (Don't Repeat Yourself). This principle is an essential axiom in programming, to minimise redundancy and promote code reuse. Adherence to this principle is crucial for developing an efficient, maintainable, readable and scalable application. DRY says that every piece of knowledge in the system should have a single authoritative and unambiguous representation. Each piece of knowledge in system development should have a single representation. The knowledge of a system is much broader than just its code. It refers to database schemas, test plans, system builds, even documentation. [5]. The DRY principle stresses the need to avoid duplication of information and logic in a software system. When a piece of code occurs more than once in an application, this can lead to divergences and difficulties in maintaining consistency between these segments. Following the DRY principle helps to reduce complexity, increase efficiency and maintain consistency across code. The DRY principle may be violated if similar code fragments are replicated in different application parts without centralizing common functionality. This can be seen in repetitive implementations of certain code sequences, causing divergence, update difficulties and increased risk of errors.

Repeating the code is not an absolute solution. Some principles encourage code repetitiveness, such as WET. WET, an acronym opposite to the concept of DRY, translates to "write everything twice" (alternatively, write every time, we like to type or we waste everyone's time). WET solutions are common in multi-tier architectures, where a developer may be burdened with, for example, adding a comment field to a web application. The text string "comment" could be repeated in a label, HTML tag, a read function name, a private variable, database DDL, queries and so on. The DRY approach eliminates this redundancy by using frameworks that reduce or eliminate all but the most important of these editing tasks, leaving the extensibility of adding new knowledge variables in one place. Kevin Greer named and described this programming principle.[4]

The context is all the more relevant as the rate of code copying in working environments is high. According to a study in Software Factories, 80 percent of code in an application is duplicated [3]. Engineers and developers avoid reimplementing already existing and optimized functionality, instead finding use in repeating existing code in other sources to solve one-off problems. From this point of view a developer needs to create code in mind with the aim that it can later be ported or reused. Such ways of thinking lead to components such as APIs or libraries. The use of a library allows the integration of standardized functionality that has

already been implemented and tested into an application. This can greatly reduce the amount of duplicate code and simplify implementation by providing basic functionality that is available and extensible. APIs provide an efficient way of communicating between different applications or components, allowing access to functionality and data without recreating existing logic. They help modularise and abstract code, reducing duplication and facilitating interoperability between systems.

1.2 Aim and objectives of the research

This research aims to analyze and highlight the advantages of following the DRY (Don't Repeat Yourself) principle in software development by evaluating two different approaches: using a library and using an API. The research will focus on demonstrating the impact of DRY compliance on efficiency and performance in application development.

The way to achieve this objective is by identifying and analysing the benefits of integrating a library into an application to reduce duplicate code. A library will be created and presented that allows elementary mathematical operations to be performed in different ways, and under certain rules. After achieving this objective, the impact on efficiency and ease of code maintenance due to using a library will be evaluated. Performance will be determined using metrics such as call speed, ease of use of methods, and processor and memory performance. This analysis will be followed by investigating the benefits of DRY compliance through the use of an API within a software application. In addition to the metrics mentioned above, the required infrastructure, differences in design mode and some of the ecosystem advantages of a microservice in which the API operates will be analyzed.

A comparison of the results obtained will be carried out to highlight the benefits and limitations of each approach in terms of compliance with the DRY principle. In retrospect, the results of using a duplicate code will also be presented.

2 METHODOLOGY

The methods of gathering data will vary depending on the metric I am planning to test. There is a particular set of information that can be gathered and can also be relevant for the problem discussed. Some of the important metrics that should be put into comparison are regarding time efficiency and lines of code, since the application of this particular design principle should bring down the complexity and number of call made for achieving one task. Also, since the comparison is between two similar approaches that are trying to achieve the same goal, it is important to set two checkpoints on where the call starts, and when it ends. Since the purpose of this application is to receive data processed after calling it through an interface, the first checkpoint will be the call of the methods we are trying to measure, and the moment they are retrieved from the processor. To see the full scalability potential, methods call will have the complexity scaled in such a way that it would take more time for the machine to process. The time measurements will be logged internally with java.time package [1].

2.1 Description of methods and techniques used for data collection and analysis.

Lines of code is the most relevant metric for the chosen design principle since its purpose is to not repeat code unnecessarily. For this metric, I am planning to use a specific Java package called Java Lines of Code Counter that is meant to recursively access all the .java files from the source code and sum up the lines. This can give an overview of how many lines of code would be added if a new source is creating the same logic instead of the already existing ones, nicely packed in a separate source.

For CPU and memory usage I plan to use IntelliJ Profiler. The IntelliJ Profiler represents a user-friendly yet robust tool designed for profiling CPU and memory allocations. It merges the capabilities of two widely used Java profilers, namely JFR and Async profiler.

Its primary emphasis lies in user-friendliness, enabling users to initiate profiling effortlessly with a few clicks, eliminating the need for intricate configurations. While featuring advanced functionalities, IntelliJ Profiler streamlines day-to-day development tasks with its intuitive interface.

As an integral component of IntelliJ IDEA Ultimate, IntelliJ Profiler seamlessly attaches to processes through a single click, allowing seamless navigation between snapshots and corresponding source code. Its additional functionalities, such as the differential flame graphs, facilitate a visual assessment of various performance approaches, providing swift and efficient

insights into runtime operations. In terms of fiability and scalability, the possible tests consist in checking the load the application can handle on different instructions it is sent to it.

2.2 Presentation of the selection criteria for the design patterns and anti-patterns analysed

One of the ways to address the DRY principle is by creating a library to encapsulate all the repeating methods.

Libraries can effectively address the Don't Repeat Yourself (DRY) pattern by encapsulating reusable functionalities or common code logic. When developers identify recurring patterns or functionalities within their codebase, they can extract these elements into a library. By doing so, they centralize and abstract the shared functionality, enabling other parts of the codebase to call upon and utilize these functions, classes, or modules.

This approach promotes code reusability and simplifies maintenance. Instead of duplicating the same logic across multiple areas of the code, developers can employ the library, thereby reducing redundancy. Any updates or improvements made to the library automatically propagate across all parts of the codebase that utilize it, ensuring consistency and reducing the chances of errors caused by inconsistent implementations.

Libraries act as a repository of reusable code components, enhancing efficiency, readability, and maintainability while adhering to the DRY principle by preventing unnecessary duplication of code.

An API (Application Programming Interface) can help mitigate the Don't Repeat Yourself (DRY) pattern by providing a standardized interface or set of functionalities that can be accessed and reused across various parts of a software system or different applications.

Here's how an API can address the DRY principle:

Centralized Functionality: APIs encapsulate specific functionalities or services, allowing multiple parts of an application or different applications altogether to access them. This prevents the need to duplicate code for the same functionalities across various components.

Reusability: Instead of recreating the same logic, an API allows developers to make calls to endpoints or methods provided by the API, enabling code reuse. This reduces redundancy and maintains consistency in how specific tasks or operations are performed within an application or across multiple applications.

Consistency and Maintenance: With an API, changes or updates to functionalities can be made within the API itself. This ensures consistency across all areas that rely on the API, reducing the chances of inconsistencies that can arise from duplicated code that needs to be individually updated.

Modularity and Scalability: APIs promote a modular approach to software development, allowing teams to build and expand applications by integrating various APIs. This modular structure supports scalability and agility in development, as new features or functionalities can be easily added by leveraging existing APIs.

Interoperability: APIs facilitate interaction and data exchange between different systems, languages, or platforms. This interoperability enables diverse applications to communicate with each other, leveraging shared functionalities without the need for duplicative implementation.

In summary, APIs adhere to the DRY principle by offering a consolidated and standardized interface for functionalities, reducing code duplication, ensuring consistency, promoting reusability, and facilitating scalability and interoperability across different parts of a software system or between various applications.

3 ANALYSIS OF DESIGN PATTERNS

There are multiple design patterns that already respect the Don't Repeat Yourself principle. Most of them are described in Design Patterns Elements of Reusable Object-Oriented Software, where the design pattern is already described like so "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [2] The essence of design patterns, as presented in the book, emphasizes creating reusable solutions to common problems in software design. By employing design patterns, developers can encapsulate commonly used structures and behaviors in a modular and reusable manner.

The "Don't Repeat Yourself" (DRY) principle in software development aims to minimize repetitive information that might undergo changes. Instead, it promotes using abstractions less likely to change or employing data normalization to prevent redundancy from the outset.

The principle is defined as ensuring that each piece of knowledge within a system has a single, clear, and authoritative representation. Originally articulated by Andy Hunt and Dave Thomas in "The Pragmatic Programmer," they extend its application broadly, encompassing database schemas, test plans, the build system, and even documentation.

Successful application of the DRY principle ensures that modifying one element of a system doesn't necessitate changes in unrelated parts. It also ensures that logically related elements change uniformly and predictably, maintaining synchronization. Thomas and Hunt advocate for methods, subroutines, code generators, automatic build systems, and scripting languages to uphold DRY across various system layers.

When it comes to solutions, two of the more advanced architectural solutions to avoid repeating of code were libraries and APIs. In the realm of computer science, a library refers to a compilation of persistent resources utilized by computer programs, predominantly in software development. These resources encompass configuration data, documentation, help files, message templates, pre-existing code snippets, subroutines, classes, values, or type specifications. In IBM's OS/360 and its subsequent versions, they are termed as partitioned data sets.

Furthermore, a library constitutes a set of behavior implementations, articulated in a specific programming language, featuring a well-defined interface through which the behavior is activated. For example, individuals intending to develop a higher-level program can utilize a library to execute system calls instead of repeatedly creating these calls from scratch. Additionally, the implemented behavior is made available for reuse by multiple independent programs. A program accesses the behavior provided by the library using the language's inherent mecha-

nism. In a straightforward imperative language like C, invoking the behavior from a library involves using C's standard function-call. The distinction between calling a library function and calling another function within the same program lies in the organizational structure of the code within the system.

Libraries are core components in old an new languages as of today. Libraries were one of the main gateways of sparring low level function from high level functionality by encapsulating repeating basic functions like printing or reading files. Standard C library for example, hides such complexities and lets the developer create more complex architecture by not bothering to rely on self made repeating code.

The history of APIs is also rich and came as a fast solution for repeating code structures. Programming languages, software libraries, computer operating systems, and computer hardware all feature APIs. While APIs have their origins in the 1940s, the actual term did not surface until the 1960s and 1970s. Presently, the term "API" commonly denotes web APIs, enabling communication between internet-connected computers. The evolution of APIs has notably contributed to the surge in microservices' popularity, where loosely connected services are accessed through public APIs [6].

Some of the most popular and used APIs at this moment are web APIs that put abstract methods or preprocessed organized data behind a facade that can be easily accessed through internet via an endpoint. APIs can be categorized by different criteria, by protocols or by use case. One of the most known APIs are the formerly known Twitter API. Twitter boasts a diverse range of APIs accessible to developers aiming to construct applications interacting with the platform. Serving as one of the most widely used social media platforms, Twitter stands as an ideal hub for businesses seeking to promote their products and services. Through Twitter's Ads API, enterprises gain access to a tool enabling the creation and administration of advertising campaigns on the Twitter platform. This API empowers businesses to craft and personalize their ads, target specific audiences, and monitor campaign performance in real-time.

For instance, tools like Buffer, designed for social media marketing management, enable users to schedule their Twitter marketing activities. These tools rely on Twitter APIs to extract analytics pertinent to their users. The Buffer dashboard showcases crucial Twitter metrics, encompassing tweets, replies, retweets, clicks, and more.

4 EFFICIENCY AND OPTIMISATION

Before analyzing the improvements, this approach needs to first setup a baseline. For this example, I implemented a basic series of arithmetic operations made in such ways that the code is repetitive, it goes through all the operations multiple times and have a level of complexity to make the improvements visible. From the first iteration from the raw code, the operation run in 1.185 seconds with a 11% load on the processor and 46Kb of memory used. The code is spread along 73 lines of code, with no logic separation and cryptic meaning of variables and meaningless naming conventions such as "x" and "y" for all the factors used in the operations. If one of the operation would have a necessary change there would be an indefinite amount of places where the code needs to be changed making it harder to maintain. Testing the code would be hard considering the tests could not be granularized. Each test would need to check different combinations of the same operations, instead of one operation in particular.

Using a library keeps the same average time of execution and CPU and Memory load, differing by the serial monolithic version in other aspects. The lines of code are reduced by 81 percent in the main caller application and by 43 percent in total considering both the library code and the main function. The level of complexity is lowered since classes of the library as well as function benefit from proper naming of used variables, documentation on the purpose of each function, granularization and separation of concerns. The purpose of the code is clearer in the main function as well considering that instead of having a series of fors doing the same operation over and over, we have method calls like "multiplication" or "division". Scalability increases, since no matter how many times and where the method is called, there is only one place that needs to be changed if any change is needed. It is N times faster for a change to be done in a library format comparative with the serial code, considering N is the number of usages of the method in the main code. Tests can be granularized and there can be one test per method, making it easier to debug. The library version of the application can be packaged as one container and deployed in a monolithic manner.

Using an API increases the time of execution by 37 percent ran locally in the same network. The startup of the API application is consuming more resources of CPU and Memory reaching up to 23% percent of load in total due to extra libraries it needs to load for exposing the endpoints, and the separation of concerns inside the API in a Model View Controller manner. This load also takes into account the fact that the two applications are running in different instances of spring. From a deployment point of view, these are two separate applications deployed separately in different containers. Scalability performances, as well as testing, are similar to ones of the library. One more difference between the last two approaches is in the number of lines of code. The base code for an API reaches up to 300 lines of code, due to necessary tools in serialisation, API endpoint management and separation between objects,

data transfer object controller and service. This structure is a pattern that lets the developer to easily build upon. This approach also enables separation of services into microservices.

5 COMPARATIVE STUDY

The library gives us the luxury of time and resource efficiency, while still keeping the clean code principles intact, as well as respecting the Don't Repeat Yourself principle. A library is most likely used in such cases for efficiency. One program is efficient when it comes to integrating libraries for not repeating code if the number of needed libraries is not to big in cooperation with the purpose of the application. Loading too many libraries in one application may lead to different problems. Increased Memory Usage: Each library adds to the memory footprint of the application. Loading numerous libraries can consume significant memory resources, potentially leading to higher memory usage, which might affect application performance. The more libraries you include, the more code needs to be loaded and managed by the Java Virtual Machine (JVM). This can increase the startup time of the application and might also impact its overall execution speed due to the additional processing overhead. Incorporating numerous libraries may also complicate the codebase. Managing dependencies, ensuring compatibility between different libraries, and keeping them up-to-date can become challenging. This complexity might result in longer development cycles and increased maintenance efforts. Different libraries might have dependencies on specific versions of other libraries or conflicting requirements. This can lead to compatibility issues, version conflicts, or unexpected behavior in the application, causing bugs that are hard to identify and resolve. Including multiple libraries can significantly increase the size of the application when deploying it. This might be a concern, especially for applications intended for deployment in resource-constrained environments. Using numerous libraries can potentially increase the attack surface of the application. If any of the included libraries have security vulnerabilities, it poses a risk to the overall security of the application.

The API is a slower and bigger version for small applications due to its multiple prerequisites and big codebase comparisons. It instead opens up a series of opportunities for the developer to extend and improve its application while also respecting the Do not Repeat Yourself principle. An API can fulfill other purposes than processing on different machines or systems. They can also have the ability to store the history of data obtained after a heavy computational series of operations. Requests can be cached for increased speed and performance. Although it is exposed to multiple security concerns and is more error-prone due to the separation of the logic in another system, it allows developers to come up with more ways to handle unexpected errors and attacks by protecting its own system instead of the entire codebase as it would be the case for using a library. An API opens the opportunity to not repeat the same code between multiple instances of the same codebase importing dynamic or static libraries.

6 ANTI-PATTERN IMPACT

Considering opportunities that the two presented variants are opening, using anti-patterns would bring the application in a tougher spot to build upon, extend or scale. Repeating the code over and over means in this scenario that big chunks of useless code are polluting the main application, bringing the program to a state where further developments are close to impossible to be done. For my test case, I considered a series of changes that I wanted to do for the code to measure the time and effort it would take to add it. I split the changes in four categories: method internal, structural, order of operations and version migration. Changes internal to methods would be, in general, easily implemented inside the one method that holds the overall logic. By not respecting the Don't Repeat Yourself principle, changing logic from one internal method would mean changing logic throughout the entire project wherever that method is used. This would transform one change into multiple, depending on the size of the application. This approach is error-prone since one can miss to do the changes across all places where the same code is used. Adding upon one method means adding the same code in multiple places. For structural changes in the application given as an example, I changed the context of the running operations. That means changing the overall structure of the main thread of the application by modifying global and environmental variables. These kinds of changes do not have any negative impact on a monolith application since the entire logic is encapsulated in one function, this instead would be a liability for an application splitter in microservices where secrets, ports, and other variables can influence other systems without knowing before deployment. Changing the order of operations within an application written in one single method is, just as well, a breaking change that would have to be done. Having logic separated in methods would give us a flow and order of operation overview before changing anything. In case there is method interdependency, we can easily group methods together and safely change the order of the methods around them. In a main function with repeating code changing the order of operation means to first set the boundaries of where logic starts and ends, decoupling variables that are linking two areas of logic, and recoupling with the existing code. Instead of moving method calls around, one developer is forced to move chunks of code, leading to bigger changes, harder to review and test.

7 CONCLUSIONS

The Don't Repeat Yourself principle stands as a foundational principle in software development, advocating against redundancy and duplication of code or information. It's considered an anti-pattern to be avoided due to several compelling reasons. Firstly, redundancy in code increases the likelihood of errors and inconsistencies. Moreover, maintaining redundant code is resource-intensive and time-consuming. Developers spend unnecessary effort maintaining and synchronizing similar pieces of code, resulting in decreased productivity and increased complexity within the codebase. DRY promotes efficiency by advocating for reusable solutions to common problems. Libraries and APIs emerge as the ideal solutions to combat this anti-pattern. Libraries encapsulate reusable functionalities or code segments, allowing developers to abstract and centralize commonly used logic. This approach enhances code reusability, readability, and maintainability by eliminating duplication and promoting modular code structures. Similarly, APIs provide standardized interfaces for interacting with functionalities or data, allowing different parts of a system or even different applications to communicate effectively. By offering a consistent way to access and manipulate functionalities or resources, APIs contribute to reducing redundancy and promoting a more streamlined and cohesive development process.In summary, the avoidance of the DRY anti-pattern is crucial for maintaining a clean, efficient, and manageable codebase. Libraries and APIs serve as effective remedies, facilitating code reuse, reducing redundancy, and enhancing overall software quality by encouraging modular, reusable, and maintainable code structures.

BIBLIOGRAFIE

[1] https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html.

[2] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[3] Cook Kent John Wiley Sons Greenfield, Short. *Software Factories*. 2004.

[4] Alex Papadimoulis. *The WET Cart*. 2011.

[5] Bill Venners. Orthogonality and the dry principle a conversation with andy hunt and dave thomas, part ii.

[6] Laura Wood. Global cloud microservices market (2021 to 2026). 2022.