UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Raido Aunpuu

Analysing the Effects of Dependencies and Abstractions in Software Development

Bachelor's Thesis (9 ECTS)

Supervisor:
Dietmar Pfahl, PhD

Tartu 2025

Analysing the Effects of Dependencies and Abstractions in Software Development

Abstract:

The aim of the thesis was to investigate the impact of dependencies and abstractions in software development. A small part of an already existing application was rewritten using guidelines that are developed in this thesis. The new application was compared with the existing one and the pros and cons of such an approach were discussed.

The thesis introduces the completed guidelines and the process which was used. In addition, the development of an application based on the guidelines is described, which provides a practical example of how to apply these instructions.

The purpose of the guidelines was to simplify application development and maintenance. The final guidelines mainly emphasize the need to carefully design abstractions and avoid relying on libraries and other external software.

Keywords: software development, maintainability, abstraction, dependencies, guidelines

CERCS: P175 Informatics, systems theory

Abstraktsioonide ja sõltuvuste mõju analüüsimine tarkvaraarenduses

Lühikokkuvõte:

Töö eesmärk oli uurida teekide ja abstraktsioonide mõju tarkvaraarenduses. Selleks kirjutati väike osa olemasolevast rakendusest ümber juhiste järgi, mis valmisid töö käigus. Uut rakendust võrreldi olemasolevaga ning arutleti sellise lähenemise plusse ja miinuseid.

Töös tutvustatakse valminud juhiseid ning loomisprotsessi. Lisaks kirjeldatakse juhiste järgi valminud rakenduse arendamist, mis pakub praktilist näidet, kuidas neid juhiseid rakendada.

Valminud juhiste eesmärk oli lihtsustada rakenduse arendamist ja hooldamist. Lõplikud juhised rõhutavad peamiselt vajadust abstraktsioone hoolikalt disainida ning vältida teekidest ja muudest välistest tarkvaradest sõltumist.

Võtmesõnad: Tarkvaraarendus, hooldatavus, abstraktsioon, sõltuvused, juhised

CERCS: P175 Informaatika, süsteemiteooria

Contents

1. Introduction

As applications grow in complexity, developers often have to make decisions concerning abstractions and dependencies which can significantly impact both the development process and long term maintainability of the systems. Abstractions can be seen as a way to generalize and eliminate unnecessary information [1]. Dependencies offer a way to rely on some ready made abstractions, this creates the challenge of choosing the right abstractions to depend on [2]. It is not easy to find the balance between leveraging existing resources and maintaining the required amount of control over application architecture. Using dependencies usually comes with the loss of control over the abstractions implementation. This loss of control can lead to missed opportunities for the design of the application, which in turn can lead to the application becoming harder to develop and maintain. On the other hand, if no dependencies are used then it raises the concern of wasting time and effort on ”reinventing the wheel”. Though this does open up possibilities which might never have been explored otherwise. This thesis aims to explore these dynamics by investigating the effects of dependencies and abstractions on software development practices.

This thesis focuses on creating a set of guidelines which detail when to use a dependency and how abstractions should be designed. The goal of the guidelines is to simplify the development and maintenance of software. Alongside the guidelines a rewrite of an existing application is developed. Since the original application is quite large then the rewrite focused mainly on providing the architecture needed and only a small part of actual features were reintroduced. The development of the guidelines happens iteratively where a new version of the application is written and then the guidelines will be adjusted with insights gathered during the development.

The thesis is split into four main sections. The first one gives an overview of abstractions and dependencies regarding the context of this thesis. The existing application which will be rewritten is also introduced in this section. The second section explains the method which is used to develop the guidelines. The third shows the results of the rewrite and the forming of the guidelines. The last one compares the existing application to the new one which was developed during this thesis.

2. Background

In this section, I explain the role of abstractions for software engineers. It explores the types of abstractions and how these can affect the complexity of software. The idea of seeing dependencies as abstractions is introduced as a way to show how similar these two concepts actually are. In addition, I briefly describe the software application my team is working on in the company I work at. This serves as the context in which the ideas developed in this thesis project are implemented. It will be the reference to the new application which will be developed by following the method of this thesis. Due to both of them containing confidential information then both will remain closed source, but both will have the relevant details described.

2.1 Abstraction in software development

Abstractions in computer science are about hiding information, usually by ignoring the features that are not of importance [2]. To understand the complex reality with which software has to interact, one needs to generalize and eliminate detailed control where it is not necessary for the current context [1].

An example of different levels of abstractions can be seen in the internet protocol suite which is defined in RFC 1122 [3] and RFC 1123 [4]. It consists of four different layers that are all responsible for their own task and serve to simplify the logic for the layer that is on top of it and hide the complexity of the layer that is beneath it. Most of the applications built today interact with the application layer, for example the Hypertext Transfer Protocol (HTTP) [5] is an application layer protocol.

Colburn et al. [2] explain that technology has evolved and now allows higher-level abstractions to be designed on top of old structures. Currently programmers can use a programming language to write programs in a textual way that abstracts away how the text will get translated into electronic signals by the computer. When designing software the challenging part is to choose the right libraries and thus abstractions to rely on. Design patterns provide guidelines on how to organize the abstractions in a way that promotes reuse and changeability. But not every design pattern fits for every use case and abstractions that implement the design patterns need to be thought out well.

The amount of tools that provide varying levels of abstractions has increased over the years, which has made the decision of what tools to use a difficult one. The challenges of choosing the right level of abstraction are explained in this paragraph relying on the following article [1].

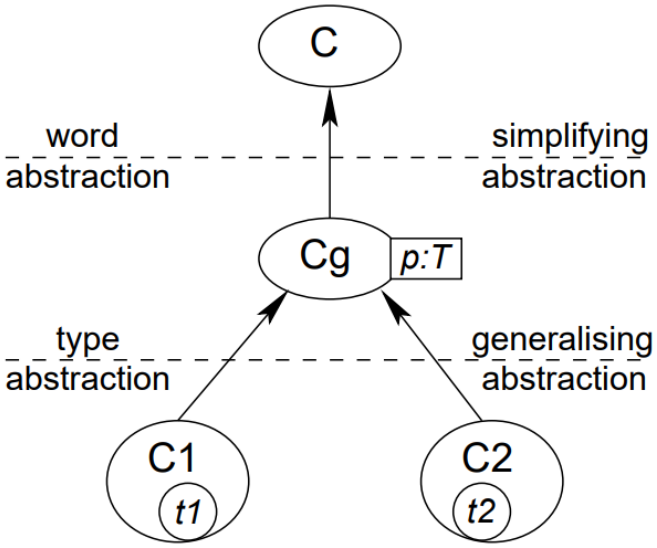

Figure 1. The two types of abstraction [1].

The challenges can be explained by exploring the relationship between abstractness, specificity and complexity. Abstraction is used as a tool to reduce the information a component holds to reduce complexity, but what is often not understood is that not every abstraction results in the reduction of complexity. The article introduces two types of abstraction: generalising and simplifying. The generalising abstraction finds something common between components and generalises them by letting the differing parts be injected into the newly made abstraction. In Figure 1, this is illustrated by taking components C1 and C2 and creating a new one called Cg which makes the differing parts (t1 and t2) be a parameter p that takes in anything of type T. Now the newly made abstraction can be reused, but the complexity is not necessarily reduced. When looking at a function then this approach reuses the common logic but introduces a new parameter and possibly a new type which can represent both t1 and t2 (type abstraction). The simplifying abstraction aims to make the component more specific. For example by removing a parameter from a function and just setting a value for it, this makes the function less complex and easier to use. In Figure 1 this is shown as the step where Cg has its parameter p removed, this means that some fixed value was set for it in C and thus the value/word was abstracted away. This means that a high level abstraction that resulted from generalising might be really flexible, but it does not always result in loss of information. The same logic is still there and the user of the abstraction still has to make a decision when passing in the data via the new parameters. This can be an example of a too high level abstraction that is deemed “too abstract” and thus is

not simpler to use. The solution might be to make it more specific and remove some parameters, but the parameters or in other words the features that will be lost must be chosen carefully.

In conclusion abstractions allow to hide complex logic behind some easier to use applications programming interface (API), but usually this means that there need to be some compromises, there have to be some decisions that are made by the one who made the API about what is important and where the user of the API probably won't need fine control of the underlying features. This means that before starting to depend on an external library one should evaluate if the offered abstractions are fitting for the software that is being developed.

2.2 Dependencies as abstractions

This section focuses on dependencies that are between two modules from the same software or a result of using code someone else has written as an imported package. These will be called internal and external dependencies from here on. Both mean reusing some logic, but from the whole project's point of view external dependency also means loss of control of the implementation with the upside of speeding up development. The logic that is being depended on can also be viewed as an abstraction since the fine details of the implementation might be hidden from the user of said logic. External dependencies can thus be seen as abstractions that's implementation is either of no concern to the project or the implementation is just what is needed. Either way, the control of the implementation is handed to the author of the package.

Possible problems with external dependencies:

- The assumption that the details of the implementation do not matter turns out to be false. For example if there are performance problems, which results in the need to either read and understand the source code to find some possible optimizations or write a new implementation from scratch thus making it an internal dependency.
- The exposed API forces its own patterns that do not align with the rest of the project. This is not always a problem but many of these combined cases can lead to having multiple design patterns used and thus hurt readability.
- The author of the package could have malicious intent and start spreading malware. An example is the xz utils backdoor [6] which is known as CVE-2024-3094. A contributor managed to hide malicious code in the project which is depended on by other software and ending up in a possible backdoor in sshd [7].

The first two points are the result of loss of control over the abstraction and the third point is the result of trusting someone else to provide the code.

Possible problems with internal dependencies:

- Fewer projects use the implementation and thus there is a higher chance there might be problems such as logic errors or bad performance.
- Higher initial cost of development.
- More code to maintain.

Every external dependency can be made internal by just downloading the source and using it directly, but is it sensible to do so? This can remove the security problems of using external code assuming that the code was audited. But the problem with loss of control does not vanish this easily since having control assumes that the code can be understood and changed without much effort. Since often external dependencies try to be as abstract as possible to handle all possible cases then there often is some leftover logic that is not relevant to the current project and if left as is can be called dead code. This means that reimplementing it to fit the project is needed for it to be called an internal dependency.

2.3 Context: Project Y

Project Y is a POS (point of sale) application that is currently being developed in the company I work for. The company works on business management software and has ~70 employees. Project Y has roughly 7-8 people involved in it including myself. Project Y consists of frontend and backend. This thesis will focus on the backend application which will be called X from here on.

2.3.1 Architecture of X

X is written in TypeScript[1] and has a PostgreSQL[2] database. For most of the needed architecture various dependencies are being utilized. The biggest and most influential is NestJS[3] which is a framework that provides layers of abstraction for most of the logic needed in a backend application. Here is a list of some of the features from NestJS documentation:

[1]https://www.typescriptlang.org/
[2]https://www.postgresql.org/
[3]https://nestjs.com/

- HTTP Server which is an abstraction on top of either Express[4] or fastify[5].
- For database connections it offers tight integration with Sequelize[6] and TypeORM[7]. Both of these offer abstractions for using many different databases under the hood.
- Dependency injection.
- Data validation pipes.
- HTTP middleware.

X uses a lot of what NestJS has to offer and follows the documentation to align with the best practices. The NestJS documentation overall is quite good and has examples on how to get started using the provided features. Figure 2 shows the most important layers of abstraction for X. These are the layers that the developer will interact with and needs to be aware of. The operating system is shown as the lowest level since this thesis will not explore going lower than that.

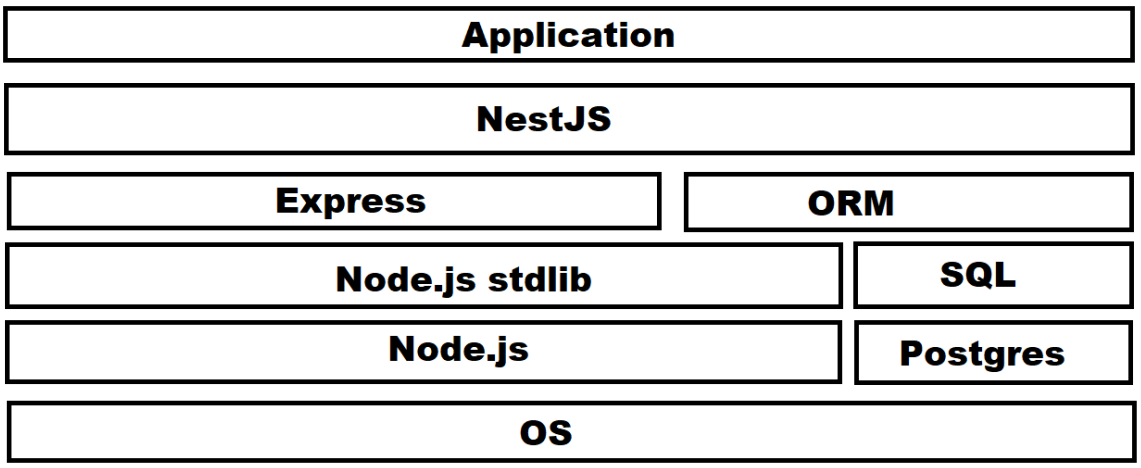

Figure 2. Layers of abstraction for X

The new application which will be called Xnew from here on will take X as a reference. Xnew will focus on the architecture which is needed to support the business logic. A small section of business logic will also be reimplemented to have a point of comparison between the projects.

[4]https://expressjs.com/
[5]https://fastify.dev/
[6]https://sequelize.org/
[7]https://typeorm.io/

This will also demonstrate how most of the rest of the business logic could be developed using the new abstractions.

2.3.2 Needed functionality of Xnew

Here is a list of features Xnew should have in order to be a possible replacement for project X:

- Accept and process HTTP requests from frontend.
- Communicate with PostgreSQL database.
- Write and store logs.
- Automatic tests.
- Data validation.
- JWT authentication.
- Error handling with proper messages.
- Deploy on a Linux machine using Docker.

The list is quite standard for backend applications. To keep the scope smaller the requirements here were chosen to require as little changes as possible to the surrounding infrastructure which is already in place for project Y. This does limit the potential of Xnew, but there is already enough room for experimentation to get some results for this thesis.

2.3.3 Quality criteria

The information presented in the following was gathered by analysing X with regards to relevant quality criteria. The focus was on quality criteria that are relevant in the long term, which in the current context means years. In the short term projects sometimes are better off with neglecting some criteria rather than missing a crucial deadline for a contract. Looking at the long term helps to eliminate having to consider these possible short term obstacles and focus on a smaller scope of software engineering.

There are different quality criteria for example ISO 25010:2011[8] and NIST SAMATE[9]. These both go quite in depth in providing a way to measure an application, but for the purposes of

[8]https://www.iso.org/standard/35733.html
[9]https://www.nist.gov/itl/ssd/software-quality-group/samate

this thesis and due to limited time there will not be a concrete framework for categorizing or evaluating the criteria. Rather simple criteria which should be easy to understand will be used and evaluated mostly by discussion. The following is a unordered list of important quality criteria for X and short explanations of why they are important.

- Readability of code - there will always be a need for new features which means having to understand old code which might need to be modified or reused.
- Infrastructure cost and complexity - costs directly drive down profits. Added complexity can mean having to hire a new person to manage it which is also added cost.
- Correctness - the application must not have mistakes since mistakes in the application can lead to huge costs and loss of clients.
- Security - The application might handle sensitive information and thus it has to be secure.
- Performance - Can directly drive up infrastructure costs. But the backend application itself does not need to be really efficient since most data processing happens in the database. The important metric for the backend is mostly latency (time it takes to initiate an action and when the response arrives) since that is what the clients will notice.

These criteria will be used later to know what to prioritize in the design and implementation phases for Xnew. The criteria can also be used to measure the improvements and discuss on the tradeoffs between X and Xnew.

3. Method

The goal of my thesis is to enhance maintainability by providing guidelines that help developers decide on when to introduce an external dependency vs when to use their own implementation. In the case of choosing to opt for the in house implementation then the guidelines will also give steps to designing these abstractions. To develop the guidelines, action research combined with the iterative Plan-Do-Check-Act approach is used. Figure 3 gives a short overview of what each step means.

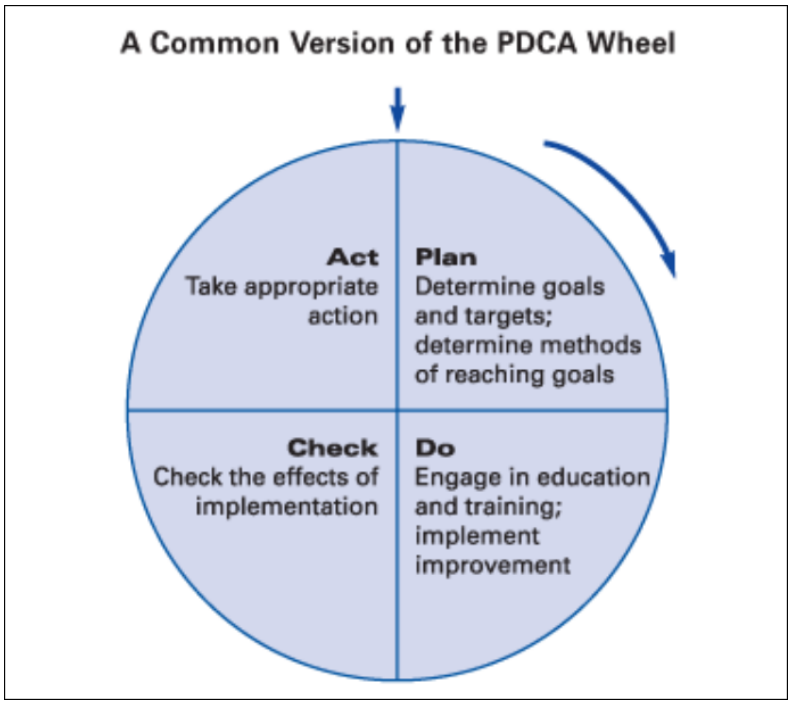

Figure 3. Plan-Do-Check-Act wheel [8].

The four steps will be repeated until the goal is reached. The input to the first iteration is version V0 of the guidelines (see Appendix 1). Guidelines V0 will be based on the findings from the background section combined with personal experiences. At the end of each iteration, an improved version Vn of the guidelines is made which serves as the input to the next iteration. I will use the guidelines to provide a proof of concept for a new version of the backend application X which will be called Xnew. Since I am also a member of the project Y development team then the development of the guidelines takes place in an action research setting. Action research is a methodology where the researchers actively change the environment and observe the results [9]. Figure 4 gives an overview of how the Plan-Do-Check-Act cycle is implemented for this thesis. The figure will be explained in more detail in subsections 3.1 to 3.4.

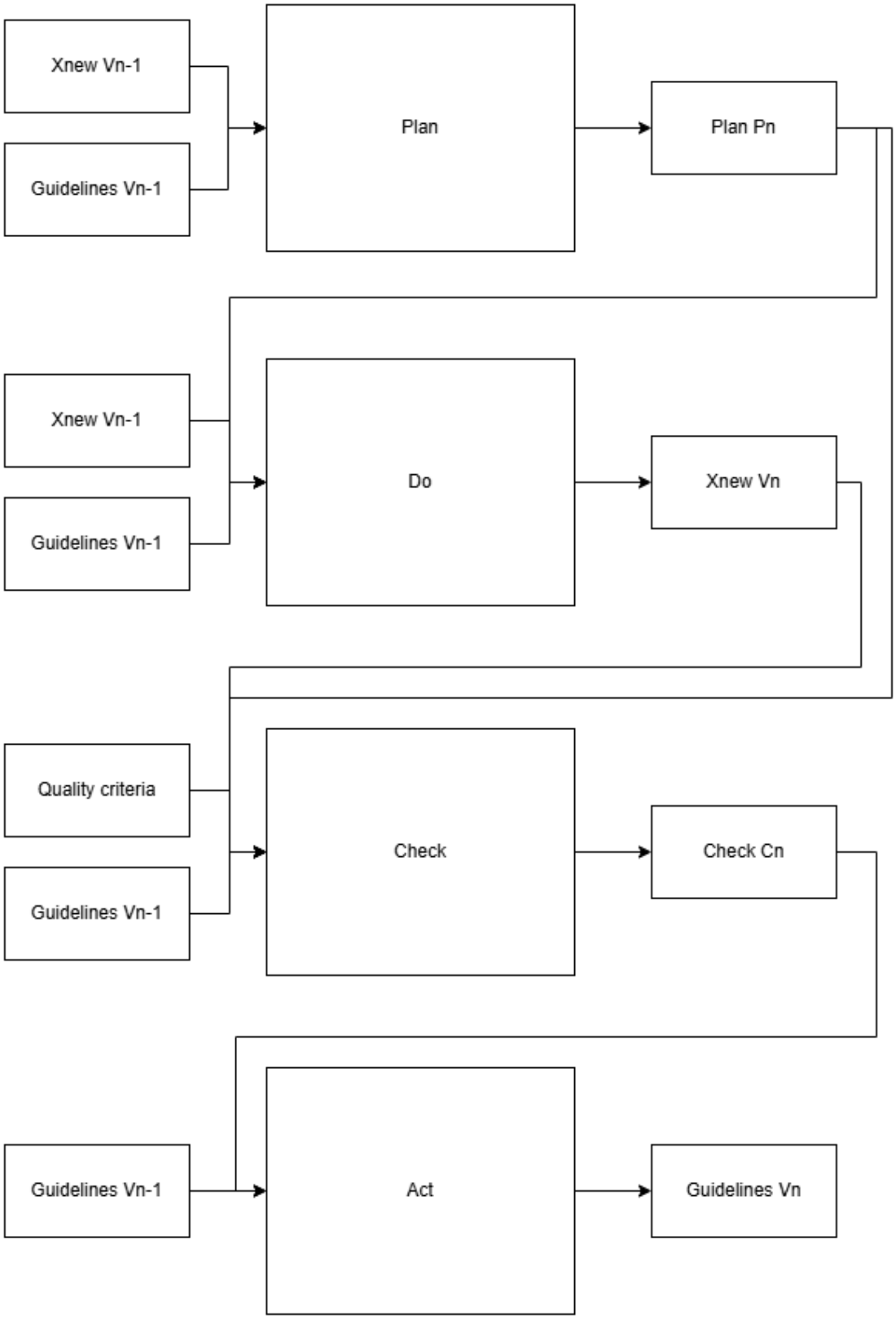

Figure 4. Plan-Do-Check-Act diagram.

3.1 Plan

Inputs:

- Xnew Vn-1 - last version of the application.
- Guidelines Vn-1 - last version of the guidelines.

Output:

- Plan Pn - the plan for this iteration.

The overall goal is to improve the guidelines by applying them to Xnew and evaluating the quality criteria to measure improvements. This step will provide a plan which details the overall changes to make to Xnew and should also provide a scope for the iteration. The plan will be based on the last versions of Xnew and guidelines.

3.2 Do

Inputs:

- Plan Pn - the plan for this iteration.
- Xnew Vn-1 - last version of the application.
- Guidelines Vn-1 - last version of the guidelines.

Output:

- Xnew Vn - the new version of the application.

The guidelines will be applied to Xnew taking into account the plan which was made for this iteration. This step will describe how the needed functionality for Xnew was implemented.

3.3 Check

Inputs:

- Plan Pn - the plan for this iteration.
- Xnew Vn - the new version of the application.
- Quality criteria of project X.
- Guidelines Vn-1 - last version of the guidelines.

Output:

- Check Cn - the findings of what and why changed positively/negatively regarding the quality criteria.

Measure the quality criteria for Vn of Xnew taking into account the scope set in the plan Pn. The scoping is mainly to help focus on a smaller set of improvements at a time and thus not have to measure everything every time. Once the quality criteria is measured then it should be analysed to find out what change in the guidelines caused it. Here is how each criteria will be measured:

- Readability of code - due to this being highly subjective it will be estimated by saying that the more logic there is that needs understanding the harder the code is to read and understand. This definition mainly helps discard the style part of readability which is not that relevant for this thesis.
- Infrastructure cost and complexity - the assumption is made that if the program needs less infrastructure then it is also less complex and probably costs less to maintain. The cost is also directly tied to performance and thus that will also be used to measure the cost.
- Correctness - the assumption is made that if a piece of code is easier to understand and thus audit then it must also be easier to spot mistakes and thus have less bugs overall. Another assumption is that the less logic a piece of code has the less surface area there is for bugs.
- Security - since security problems often result from problems with correctness then mostly the same points apply. But in addition there is the issue of package authors being potentially malicious and thus the assumption is made that the less dependencies there are the better the security is.
- Performance - will be measured by concrete measurements or if feasible then by simple argumentation.

Most of these could have been measured by other methods which offer a better accuracy and confidence for the results, but the focus of this thesis is to find spots which result in large improvements which should be noticeable without complex analysis.

3.4 Act

Inputs:

- Check Cn - the findings of what and why changed positively/negatively regarding the quality criteria.

- Guidelines Vn-1 - last version of the guidelines.

Output:

- Guidelines Vn - new version of the guidelines.

Changes to the guidelines will be made which will try to address the shortcomings found in Check Cn. After this a decision is made whether to iterate further or finalize the guidelines. If there is little room for improvement then the iterating will stop and the guidelines Vn will be determined to be the final result. Otherwise a new iteration will be started with the new guidelines Vn and Xnew Vn as the input.

4. Results

This section will describe the iterations. Each iterations section will explain what was done in each of the plan-do-check-act steps during that iteration. The resulting guidelines which are produced in each iteration can be found in the appendices section 7.

4.1 First iteration

4.1.1 Plan

Inputs:

- X - the current TypeScript application.
- Guidelines V0 (see Appendix 1).

Output:

- Plan P1.

Since the V0 of the guidelines point out to also experiment with architectural changes then I decided to start by finding possible replacements for the current programming language TypeScript.

After some exploration and research I decided to swap to Go[10] for Xnew. Go was chosen since it is a relatively fast and simple language with a good standard library which solves most of the problems without the need for a dependency.

Other candidates were TypeScript and Java. TypeScript was not chosen since it compiles to JavaScript which does have some drawbacks regarding performance and correctness. For example JavaScript numbers are all double precision 64 bit which means that there is no native integer type [10]. Besides often using more memory this also means that there is no good way for working with 64 bit integers without some workarounds. JavaScript is also notorious for its implicit conversion during various operations, for example it allows adding a number and string together without any errors. TypeScript does offer some type safety, but in practice it is not as robust as a language with static typing. In terms of the syntax and features of the language TypeScript just offered too much. The rich type system gives the opportunity to write

[10] https://go.dev/

”clever” code which turns out to be harder to understand when having to read it later. Even if the developers would agree to not use these features too much then in practice the dependencies do often force the patterns which makes this kind of code hard to avoid.

Choosing between Java and Go was harder since they both do not have any large drawbacks. Go was chosen because it just felt simpler and had more features without relying on external tooling. For example Java needs a build tool like Maven[11] or Gradle[12] to even be able to use dependencies in a sane way. Go has a module system and testing capabilities already included with the Go toolchain. Another point in favour of Go is that distributing it is simpler due to not needing to also install a Java virtual machine. Go can just be compiled for the target machine and then the binary directly executed on the target. This can also be done using Java if the virtual machine is bundled with the executable but it is not the default and is still extra friction for the developers.

Changing the programming language already means that the other dependencies couldn’t be reused. Replacements will be found by picking something that has the required functionality. The guidelines also mention cutting away layers of abstraction and favour more specific dependencies. This means that there will not be a framework like NestJS chosen for Xnew. In conclusion plan P1 contains the following:

- Use Go for Xnew V1.
- Find dependencies which provide the required functionality.
- Avoid large frameworks, prefer small specific dependencies.

4.1.2 Do

Inputs:

- Plan P1.
- X - the current TypeScript application.
- Guidelines V0.

Output:

[11] https://maven.apache.org/
[12] https://gradle.org/

- Xnew V1.

Figure 5 shows the most relevant layers of abstraction for Xnew V1. Compared to X this has the NestJS layer removed but the functionalities replaced with independent packages. Most of the packages were chosen with the goal of not having to rewrite too much from scratch. These packages are fairly reputable which seem to be common ones that others also use. Here is the list of needed functionality and package which provided it.

- Accept and process HTTP requests from frontend - Fiber[13].
- Communicate with PostgreSQL database - GORM[14].
- Write and store logs - Logrus[15].
- Automatic tests - Go standard library testing package[16].
- Data validation - go-playground/validator[17].
- JWT authentication - golang-jwt/jwt[18]
- Error handling with proper messages - Go builtin error interface using errors as values.
- Deploy on a Linux machine using Docker. This step was not done for this iteration.

Regarding the error handling Go mainly encourages writing code that uses errors as values. Exception-like semantics are still possible, but those are not widely used. Errors as values means that functions can return a simple value that contains some data if there was an error. In most cases this proved to be better than exceptions since it encouraged actually thinking about the errors and adding meaningful context at each layer of the callstack. Sadly in my opinion Go did not choose the best way to implement it since there is no good way to know what types of errors could be returned. In practice the documentation of the libraries was often enough. Since this is the first iteration this is still mostly prototyping and thus not all features were polished to meet production standards.

---

[13] https://github.com/gofiber/fiber
[14] https://github.com/go-gorm/gorm
[15] https://github.com/sirupsen/logrus
[16] https://pkg.go.dev/testing
[17] https://github.com/go-playground/validator
[18] https://github.com/golang-jwt/jwt

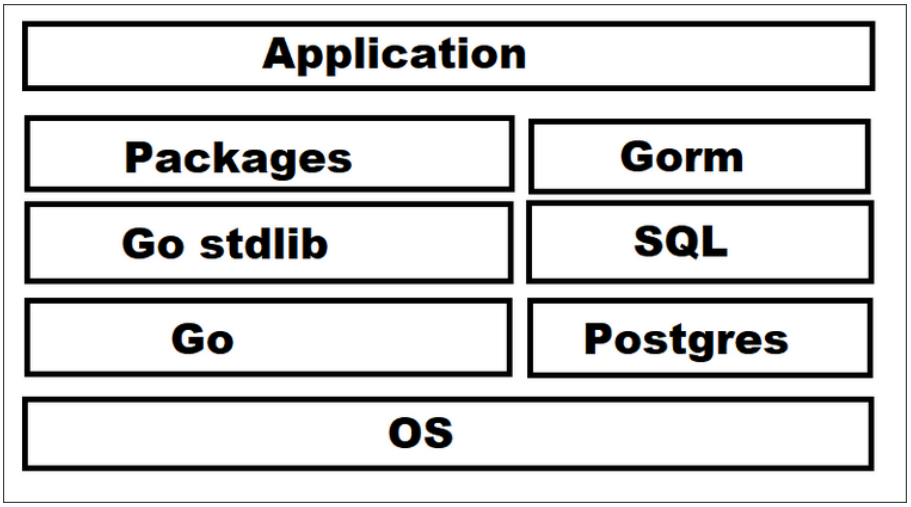

Figure 5. Layers of abstraction for Xnew V1

4.1.3 Check

Inputs:

- Plan P1.
- Xnew V1.
- Quality criteria of project X.
- Guidelines V0.

The code at first glance also seemed to be fine readability wise, but when looking at the bigger picture it was clear it could be better. These abstractions the dependencies offered seemed helpful at the start, but overall did not seem like the best solution. For example the validator package makes use of struct tags which at first seem handy and helpful, but fall apart once there is something more complicated needed. The large amount of options just means that verifying code where these struct tags are used can mean regularly diving into the documentation.

Since this time there also wasn't a large framework like NestJS tying things together, some code had to be written manually to have the dependencies work with each other. But since that small amount of code was fairly easy to write and also provided a really thin abstraction then this was not an issue.

The output of this step C1 contains:

- Readability of code - hard to say there was any meaningful improvement over X. Overall Go code in my opinion is easier to understand, but all the used dependencies should also be

understood. As was with X there still is the need to occasionally consult the documentation even for simple tasks. I felt that the problems the dependencies solved could be done with less code if it was tailored for this project. This would mean there is less logic that needs understanding and thus the code would also be easier to understand.

- Infrastructure cost and complexity - Go is easy to statically compile which means it could be distributed as a simple binary. This also has benefits when using Docker since a multi stage build can be used to have a very thin image. It is even possible to have nothing but the binary in the container. This means that the container is smaller and has a smaller attack surface also leading to better security.
- Correctness - compared to X there weren't any meaningful changes.
- Security - Xnew had about 50 total packages compared to X having over 1000. This includes packages which were required by other packages. This certainly does reduce the chance of using a malicious package, but without a long analysis it is hard to say how big the impact is.
- Performance - Go overall has better performance when compared to TypeScript. One simple example is that Go can use all the cores of the CPU [11, 12] while NodeJS mainly uses only 1-2 cores of the CPU [13, 14].

The measurements overall saw some improvement, but the main takeaway is that there is still room for improvement.

4.1.4 Act

Inputs:

- Check C1.
- Guidelines V0.

Output:

- Guidelines V1.

A new iteration will be made. What did become clear is that the swap to Go opened up a lot of new possibilities just because it had a much higher potential performance wise and required less tooling to get started. With better performance there might not be need to offload some tasks to dependencies which would use lower level languages like C. Also when the program

already performs fairly well then new features might not need complex algorithms since the performance is already acceptable. These complex algorithms could be implemented later once the need arises. The performance benefits of swapping to Go are explained in greater detail in the final comparison of X and Xnew in section 5.2.

The most important part is the foundation of the project or in this case the programming language chosen. Also the idea of avoiding large frameworks seemed to work by providing more control and room for possible experimentation. The added control mainly refers to not being restricted to the ecosystem of some framework and being able to choose from a larger pool of solutions. Doesn't matter if these are external dependencies or code written from scratch.

While reading the Go standard library source code I found that it already contains most of the features I need if I am willing to write some code from scratch. While exploring the external dependencies I used I also found that those often have so many features of which only a few interest me. But to serve all of those features the package often has its own abstractions it uses. It is often the case that the more features a package offers the more layers of abstractions it needs for its inner workings. This means that if I want to use only a small part of it and happen to read the source I still need to understand and step through these abstractions. In this case I would need to find more specialized packages but finding something that fits my needs exactly is quite hard. This means that in some cases it is probably easiest to write the functionality from scratch which is especially easier when using a language model to generate some quick prototypes. With this in mind writing most of the functionality from scratch will be explored in the next iteration.

During this iteration it became clear that the added control was mostly positive but it has to be used correctly. Just choosing some packages which have the features is not acceptable. The way those features are implemented has to be carefully evaluated and compared against the simplest solution. Where is the line of what to write from scratch and what to take in as a dependency? The next guidelines will be changed according to what I learned while prototyping in this iteration combined with relevant research.

4.2 Second iteration

4.2.1 Plan

Inputs:

- Xnew V1.
- Guidelines V1 (see Appendix 2).

Output:

- Plan P2.

Go and its standard library will be foundation for the project. Going lower than Go is probably not needed since there is not that much to gain aside from potential performance. But since the performance is already in an acceptable state in X then it can be assumed that what Go can offer is more than enough. In terms of removing a layer in hopes of reducing the amount of underlying logic that would mean either interfacing directly with the OS (skipping the standard library) or creating a custom OS for this project. Both of these expose too much irrelevant information regarding this project. The use of most of the current packages should be reevaluated since they interact with the core functionality of the application. Most of them can probably just be removed in favour of creating abstractions from scratch.

In conclusion plan P2 contains the following:

- Reevaluate most of the current packages since those interact with the core functionality of the application.
- Create minimal abstractions on top of the standard library to replace the removed packages.
- Use a dependency for the PostgreSQL connection since there is no need to rewrite that logic.

This might lead to writing too much from scratch, but for the sake of experimentation it would be interesting to find out where the limit is. In the end it is easier to just add dependencies later compared to removing them.

4.2.2 Do

Inputs:

- Plan P2.
- Xnew V1.
- Guidelines V1.

Output:

- Xnew V2.

Here is the same list as in the first iteration which shows how certain functionalities got implemented.

- Accept and process HTTP requests from frontend - Go standard library HTTP package[19]. There were some helper functions written for reading the request data from JSON and also to respond with JSON, but largely what the package offered could be used directly.
- Communicate with PostgreSQL database - GORM[20] used pgx[21] internally and thus the swap was made to use that one directly. This meant writing raw SQL for every query. This change was made because with GORM (or with any ORM) it just felt useless since I often first wrote (or thought out) the SQL and then had to figure out how to make the ORM output the same SQL. On its own this wouldn't have been enough to justify the swap to completely using raw SQL. The swap was mainly motivated by personal preferences and desire to experiment and not that much by there being something wrong with using an ORM.
- Write and store logs - a thin wrapper was written around the standard library log package[22]. The package already has features which can change the output file and a simple print function for strings. The wrapper only had to include some extra information like logging levels, default log content and format.
- Automatic tests - no changes since last iteration.
- Data validation - the types which needed validation had a simple validation method attached to them. Another alternative is to create a new type which is meant to be obtained only by using a method which validates a regular type and returns the new type. Then this new type can be used in places where the data is assumed to be validated.
- Error handling with proper messages - no changes since last iteration.
- JWT authentication - no changes since last iteration.

[19] https://pkg.go.dev/net/http
[20] https://github.com/go-gorm/gorm
[21] https://github.com/jackc/pgx
[22] https://pkg.go.dev/log

- Deploy on a Linux machine using Docker - this included writing a Dockerfile[23] to build the container. It used the official Go image[24] to build the binary and a scratch image[25] to run the binary.

Figure 6 shows the different layers of abstraction for Xnew V2. When comparing this to Xnew V1 then the packages and GORM layers are removed.

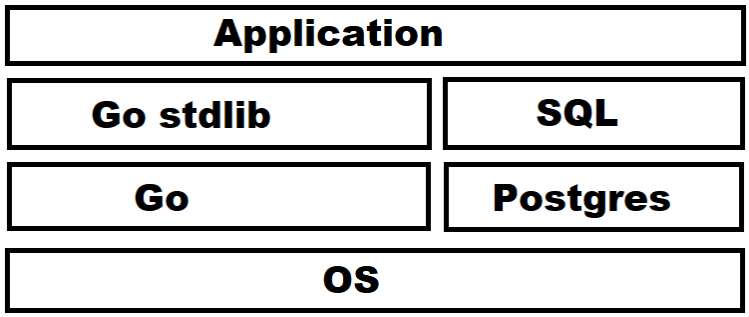

Figure 6. Layers of abstraction for Xnew V2

In conclusion most dependencies were removed since after inspecting how they worked it turned out that the needed logic was actually really easy to create from scratch. Losing GORM now meant that there was a need for a migrator for the database schema updates. This was solved by writing a migrator from scratch. The boilerplate code which had to be written to wrap the SQL to create properly typed Go functions was another issue. Creating this boilerplate was tedious and error prone and thus after some researching a code generation tool called sqlc[26] was used to generate Go functions from SQL. The functions sqlc generated were almost identical to the ones that were previously written from scratch. Code generation used to be a foreign concept since other languages like TypeScript have a powerful type system which can be utilized to create these types in a more elegant way. But I would always prefer the code generation route since it is much more readable and does not hide the important details behind some complicated syntax which is hard to understand. This can be summarized well with the following rule from the guidelines: ”Since most time is spent reading code then optimize code for readability”.

[23] https://docs.docker.com/reference/dockerfile/
[24] https://hub.docker.com/_/golang
[25] https://hub.docker.com/_/scratch
[26] https://sqlc.dev/

4.2.3 Check

Inputs:

- Plan P2.
- Xnew V2.
- Quality criteria of project X.
- Guidelines V1.

Compared to the first iteration I think that readability got improved a lot. Now there is a lot less of documentation to read and all the code is trivial and explicit which means it should be a lot easier to read. The new very thin abstractions which were made on top of Go standard library are just what the project needs and nothing more. Instead of reading documentation it is much easier to just jump into the code and read that since there just isn't that much to read. Since there were not many restrictions in place on how to structure the code I also had complete freedom to do it however made sense for the specific task. For example the HTTP handlers were the source of truth for most of the errors that the frontend could receive. This meant that all the handlers would handle all of the errors and there was no global error handler or global middleware. The main argument for this was that it is now easy to audit where and why a certain status code or response came from. This is mostly a design decision but I had full control over the following: I wrote the error messages, I decided what errors my abstractions return and what got logged immediately with a certain level and so on. In other words I could tightly integrate my logger with my own abstractions and these abstractions also had the exact error messages I cared about and would return just the errors that I wanted. All this combined made experimenting with designs a lot easier and also meant that the design for the handlers was easier to implement compared to the first iteration where this would have required understanding how, what and when do packages return errors. I still had to read how to handle the standard library errors but that was not challenging due to them usually having it well documented and also since the standard library is a lot thinner than those packages.

The output of this step C2 contains:

- Readability of code - in my opinion saw a major improvement and I am really happy with the current code and the direction the guidelines moved in.

- Infrastructure cost and complexity - no changes since last iteration, is already in an acceptable state.
- Correctness - Go standard library is probably more reliable and correct when compared to third party packages. With this in mind it can be said that correctness was also improved.
- Security - the amount of packages is even lower now. Also as with correctness I think it is safe to assume that Go standard library code is either as safe or safer to use compared to third party packages.
- Performance - no changes since last iteration.

The criteria either saw an improvement or stayed roughly the same. Compared to X or Xnew V1 there certainly aren't any large issues.

4.2.4 Act

Inputs:

- Check C1.
- Guidelines V1.

Output:

- Guidelines V2.

During the rewrites it became clear that blindly just starting to write code meant that there is a danger of making the same mistakes which others have possibly done themselves and which could be avoided by just using a package. Blindly rewriting can be called ”reinventing the wheel” and this is what is told to anyone who wants to rewrite a package from scratch. The motivation to rewrite does not come from thin air and thus the answer must be more complicated than ”just use a well maintained dependency that everyone else uses”. The key part here is that the major flaws come from not knowing how to design the abstraction, not because the implementation is made from scratch. The designs of other packages for the same abstraction should be carefully studied. With that knowledge writing a slightly modified design and implementation that is carefully tailored for the current context should be easy. Implementation can still go wrong and result in subtle logic errors, but these do not force a total redesign.

After adjustments are made to the guidelines these can be accepted as the final version. This is because Xnew V2 already showed great potential and there doesn't seem to be a lot left to

improve with the restriction of having to be roughly compatible with X. The new guidelines will also include longer explanations of what I have learned during this whole process, but the core ideas are mostly the same as V1. Not all of the ideas could be explored in this thesis due to needing a much larger refactor which would also include rethinking the whole architecture around X.

One aspect which could be explored more is code generation. Sqlc is an interesting example of how the boilerplate problem can be solved for the abstraction between Go and SQL. Perhaps this same concept could be used more widely, for example to generate the HTTP handlers from the API docs instead of generating docs from the code.

5. Comparison

This section compares Xnew V2 against X regarding the quality criteria.

5.1 Readability of code

Readability overall is really subjective to measure and largely depends on the developers and who to ask from. But due to Xnew having less dependencies and less overall logic running under the hood then it can be argued that it is easier to understand how the internals work. This increased knowledge of internals should result in the quality of code being better and thus end up with better readability. This all assumes that the chosen abstractions are good enough, but at least the potential for improvement should be better due to the extra control Xnew has over the abstractions. Another aspect is that since Xnew could learn from X and could already do some refactoring then it can also mean having better readability. But this wouldn't be due to the new design used in Xnew. All of this combined makes it quite hard to give a definitive answer to this subjective question, but overall I do feel confident that Xnew is an improvement over X mainly due to the steps taken to follow the guidelines.

5.2 Performance

Regarding performance table 1 has some rough measurements. Since performance measurements are really complicated to do thoroughly and are not a large focus of this thesis then these are the ones which were easiest to measure or ones I wanted to bring attention to.

Table 1. Performance comparison of X and Xnew

	X	**Xnew**
startup time (seconds)	5	less than 0.5
compile time (seconds)	9-11	2-3
response time (ms)	70-100	20-40
Docker image size	1.17GB	15.8MB

The startup time was measured on an already compiled application. For Xnew it meant running a native executable and for X it meant running JavaScript (TypeScript compiles to JavaScript) with NodeJS. The server was considered running once it was ready to accept and process requests. X was slow to start mainly due to how NestJS works.

The response times were measured on an endpoint which has the same business logic in both X and Xnew. The servers ran on localhost and Postman[27] was used as the client to make the request and get the elapsed time. The logic in the endpoint is minimal and does little actual work. It just has to accept the request, validate the JWT and insert the data directly into the database. For requests which need more business logic it is expected that since Go is faster overall then these differences will probably be bigger and in favour of Xnew.

Another great aspect of Go is that it can use all the cores of the CPU [11, 12] to process the requests and this is the default behaviour for the standard library http package [15] which Xnew uses. NodeJS has only one thread for the event loop which can offload some tasks like I/O and a few other ones to a worker pool where there are some other threads, but mostly all the rest of the logic runs on one thread [13, 14]. This means that NodeJS usually does not utilize more than 1-2 cores of the CPU.

In reality both application are currently in an acceptable state and thus there has been little effort put to make X better. It probably could be made better, since X has a lot more code and logic running under the hood then it is probably harder to profile and optimize. Keep in mind that since also Xnew has not had any explicit performance enhancements then this illustrates what the starting point for each of these applications would be if performance would become a concern. Since Xnew starts at an advantage it means that performance problems will probably come much later and thus wouldn't halt the development of new features for some time.

5.3 Infrastructure cost and complexity

Regarding infrastructure cost and complexity it is quite the same as with the first iteration. Due to Xnew being overall more performant then it is safe to assume that it will also require less hardware and thus result in lower cost of infrastructure. Also since modern processors especially server grade ones have more than 1-2 cores then for X it means having to horizontally scale to use these remaining cores. For Xnew it can use the cores without the need to scale horizontally and thus may not even need horizontal scaling which can help keep the complexity of infrastructure lower. This can also completely avoid having to have multiple instances running on one machine and can only deal with scaling between physical hardware. Thus having to hire someone to handle the complexities can be avoided.

[27] https://www.postman.com/

5.4 Correctness

Xnew is mostly built on top of Go standard library. With the assumption that the Go standard library is correct correctness couldn't have gotten worse. Since other Go programs also rely on the standard library, but not everyone relies on certain third party packages then it is safe to assume that the standard library is a more correct foundation. This mainly means that if something is not correct then it is probably the usage of the standard library or in the code written for Xnew. In contrast X mostly depends on third party code and thus relies on those being correct in addition to the code written for X itself. With the assumption that readability was improved for Xnew it can also be argued that the code is easier to audit for mistakes and thus is also more correct. But the caveat is that since Xnew replaces most of the third party code with the Go standard library and some wrappers then those wrappers have to be more correct than the third party code those replace. In practice since the wrappers can be quite small then it is easier to design and audit them. In my opinion this is enough evidence that Xnew is also more correct.

5.5 Security

Xnew mainly uses the Go standard library which is probably more secure and kept up to date with patches when compared to third party packages. Another easy point of comparison is the amount of dependencies. According to Docker Scout[28] the container for Xnew has 13 and X at the moment of measuring had 1555. Some of the dependencies for X were from the containers image itself but over 1000 were still from the imported dependencies being used in X. If Xnew would continue to see development it would probably get a few more dependencies, but it will probably not get anywhere close to X. In practice this meant that Xnew also saw a lot less new vulnerabilities being discovered and their severity was also lower. This is important since this means that X probably has many vulnerabilities which just have not been discovered or reported yet. For example I jumped back two months in Git history and X reported according to npm audit[29] 1 low, 3 moderate and 6 high severity vulnerabilities while Xnew reported only one medium vulnerability according to govulncheck[30]. Vulnerability analysis is also a really complicated subject and there are other aspects like the amount and credibility of different authors which is also important to factor in. Even if this is factored in the Go standard library

[28] https://docs.docker.com/scout/
[29] https://docs.npmjs.com/cli/v9/commands/npm-audit
[30] https://pkg.go.dev/golang.org/x/vuln/cmd/govulncheck

authors are probably a lot more credible than third party maintainers. In my opinion this is enough evidence that Xnew has better security.

6. Discussion

While I find Xnew to be an improvement over X, I also acknowledge that the guidelines and thus the resulting Xnew require a vastly different way of thinking which might just not work for everyone. There are occurrences where the whole team agrees that X could be simpler and improved, but there still is scepticism in the team about if the design of Xnew is the solution. Everyone mostly agrees that Xnew has better potential for quality, but the idea of starting from scratch to achieve this is not easy to justify in the context of business requirements. This criticism makes sense since the common argument is that if there exist packages that everyone else is using then why waste time reinventing them instead of using them? Another downside of Xnew is that due to the extra control it exposes to the developers it also introduces the risk of misusing this control which can end up in design and implementation flaws. While I find that the removed layers of abstraction make the code easier to understand and thus make it easier to write correct code, it is hard to prove either case. In conclusion I think that each developer should try to explore the ideas introduced in the guidelines. I learned a lot during the research and development needed for Xnew since I had to learn how the internals of the different dependencies worked and now have a much better understanding on how these tools are best used while developing X with the rest of my team.

7. Conclusion

The goal of this thesis was to create guidelines to simplify the development and maintenance of software. An application which was a rewrite of a small part of an existing application was also developed. The application was used to improve and evaluate the guidelines. The main takeaway from the final guidelines (see Appendix 3) is to design each piece of code carefully and try to minimize the use of dependencies and thus also remove layers of abstraction. The new application was compared to the existing one and the results were mostly positive, but the main concern is that the guidelines require a vastly different way of thinking which might not work for everyone. Another concern is that not using dependencies and operating at a lower level of abstraction also exposes lots of control which can be misused by less experienced developers. On the other hand the extra control allows to explore design and implementation decisions which could be just what is needed to offer a better product compared to competitors.

Future work could try to explore the area of code generation which generates source code from annotations or specifications. The advantage of generated code is that it can be easier to read when compared to metaprogramming techniques which generate the code during compilation. This thesis stopped at the standard library abstraction, but it would be interesting to find out what could be achieved by going even lower and for example creating a custom OS tailored to a backend or similar application.

References

[1] Wagner S. and Deissenboeck F. Abstractness, specificity, and complexity in software design. *CoRR* abs/1709.01304 (2017). arXiv: 1709.01304. http://arxiv.org/abs/1709.01304.

[2] Colburn T. and Shute G. Abstraction in Computer Science. *Minds and Machines* 17.2 (2007), pp. 169–184. DOI: 10.1007/s11023-007-9061-7. https://doi.org/10.1007/s11023-007-9061-7.

[3] Braden R. T. Requirements for Internet Hosts - Communication Layers. RFC 1122. Oct. 1989. DOI: 10.17487/RFC1122. https://www.rfc-editor.org/info/rfc1122.

[4] Braden R. T. Requirements for Internet Hosts - Application and Support. RFC 1123. Oct. 1989. DOI: 10.17487/RFC1123. https://www.rfc-editor.org/info/rfc1123.

[5] Fielding R. T., Nottingham M., and Reschke J. HTTP Semantics. RFC 9110. June 2022. DOI: 10.17487/RFC9110. https://www.rfc-editor.org/info/rfc9110.

[6] Standards N. I. of and Technology. CVE-2024-3094. Accessed: 2025-01-19. 2024. https://nvd.nist.gov/vuln/detail/CVE-2024-3094.

[7] Ubuntu. CVE-2024-3094. Accessed: 2025-01-19. 2024. https://ubuntu.com/security/CVE-2024-3094.

[8] lean.org. Plan, Do, Check, Act (PDCA). Accessed: 2025-05-10. https://www.lean.org/lexicon-terms/pdca/.

[9] IxDF I. D. F. .-. What is Action Research? Accessed: 2025-05-12. Sept. 2016. https://www.interaction-design.org/literature/topics/action-research.

[10] developer.mozilla.org. Numbers and strings. Accessed: 2025-05-04. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Numbers_and_strings.

[11] pkg.go.dev. runtime package documentation. Accessed: 2025-05-04. https://pkg.go.dev/runtime.

[12] go.dev. runtime HACKING. Accessed: 2025-05-04. https://go.dev/src/runtime/HACKING.

[13] nodejs.org. The Node.js Event Loop. Accessed: 2025-05-04. https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick.

[14] nodejs.org. Don’t Block the Event Loop (or the Worker Pool). Accessed: 2025-05-04. https://nodejs.org/en/learn/asynchronous-work/dont-block-the-event-loop.

[15] pkg.go.dev. net/http package documentation. Accessed: 2025-05-04. https://pkg.go.dev/net/http#Serve.

[16] Feldman R. Hybrid-Level Programming. Accessed: 2025-04-21. https://www.youtube.com/watch?v=ug-KuDlMTYw.

[17] Martin R. C. Clean Code - a Handbook of Agile Software Craftsmanship. Prentice Hall, 2009.

[18] Spolsky J. The Law of Leaky Abstractions. Accessed: 2025-04-21. https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/.

Appendices

1. Guidelines V0

The overall idea is to remove layers of abstraction. This will probably result in also using less dependencies since those usually contribute to extra layers of abstractions. The following are steps which should be done:

- Find layers of abstractions which could potentially be removed.
- Find dependencies which are more specific and which have fewer dependencies themselves.

There shouldn't be anything rewritten from scratch if there already is some dependency which could be used. Dependencies should either be replaced or removed as a result of a whole layer of abstraction being removed rather than rewriting something. Experimentation is encouraged throughout the process to avoid getting stuck with suboptimal solutions which can also be viewed as being stuck in a local minimum. This means trying large architectural changes.

2. Guidelines V1

The overall idea is to remove layers of abstraction. This will probably result in also using less dependencies since those usually contribute to extra layers of abstractions. The following sections explain the core ideas.

2.1 Design from the ground up

The first step for a project is to find out the baseline layer of abstraction which will be the lowest foundation for the project. Depending on the needed features find the lowest layer of abstraction which covers the needs. As Richard Feldman explained in his talk ”Hybrid-Level Programming” the layers of abstractions can limit an applications potential speed. In his talk he also explained that there really is no need to have so many layers of different abstractions. It is often enough to just write a thin high level abstraction on top of the low level one which maximizes performance and control but does not compromise on usability for the users of the high level API [16].

The benefits of choosing the lowest possible layer might not only show in performance but also help keep complexity away. If there is less logic then there is less to learn and less possible spots of failure. For example communicating over the network it can sometimes be better to read the data straight from the socket, cast the raw bytes into a type and optionally do some integrity checks. In contrast to the traditional approach of using HTTP which in itself adds lots of complexity and some performance overhead due to the needed string parsing to extract the data. Also with HTTP the data is usually sent using JSON which also means having to have some parsing which ends up in performance overhead and again added complexity. In some cases it might be easier to just find out what is actually needed and create the simplest logic that handles it. Even relying on already existing libraries for HTTP might not pay off in the long run due to the added complexity. Since in this case learning how to use the HTTP library correctly can be harder than learning the custom protocol which only does one simple thing. One could argue that a developer needs to learn HTTP only once but it is still mental overhead. The goal of this design is to keep mental overhead to a minimum.

In conclusion the principle of designing from the ground up means to identify the right abstraction from which to start from. It also includes finding the minimal logic needed to implement the features. There should also be some time dedicated to finding out if the features even need to be that complex and maybe changing the design can lead to reducing the needed amount of logic. The goal should be to reduce mental overhead for the readers of the code and keep everything

as simple as reasonably possible. The foundation should be used as a base on which future layers of abstraction can be built upon if the need arises. This whole ordeal can mean inventing something new and this behaviour should not be feared.

2.2 Designing abstractions

After the baseline abstraction is chosen then the application may suffer from having too much information exposed about the inner details. The natural step is to start creating functions and/or classes which encapsulate similar logic. In other words starting to create generalizing abstractions to gather similar types of functionality together and simplifying abstractions to hide the details which are not needed. The following is a list of rules for designing abstractions:

- Prefer creating abstractions from scratch. This means having full control over them which means they can be made to fit perfectly with the other abstractions.
- Since most time is spent reading code then optimize code for readability [17].
- Avoid creating abstractions which are used only once. It is hard to know what the abstraction should look like with only one use case.

These rules mostly apply to the logic which is the core functionality of the application. This is the logic which has the most effort put into it and where bad decisions can have the most negative impact. Usually a rule of thumb is to always ask what information is relevant for the current context of the code. Avoid hiding information which is actually relevant for the users of the abstraction. This means that some information is hard to hide since the problem just has that complexity in it. Trying to hide the complexity behind some abstractions only means that it will start to leak at some point. A better approach would be to try to redo the design which added the underlying complexity. If the design can't be optimized then the problem just has that complexity and the complexity should just be accepted. The layers of abstraction should be kept thin and minimal. Every layer adds another mental overhead when there is a need to constantly jump between layers during development.

2.3 Choosing dependencies

The following are rules to follow when deciding on if to use a dependency and what dependency to use.

- Consider using a dependency for logic which is not relevant to the core of the application. This means that the details of the implementation are not that relevant and thus it can be acceptable to use a dependency to cover that functionality.
- Use a dependency if the required logic is too complicated to write from scratch.
- Use a dependency if it is exactly what is needed and fits the goals of the application.

One example of an acceptable dependency for backend applications can be the database. Databases are typically too complex to implement from scratch. However, they manage the complexity of reliable data storage, and they're relatively easy to isolate from business logic since they serve a specific purpose. Though often databases are a leaky abstraction since sometimes the engines internals do become relevant for performance critical queries.

3. Guideline V2

The overall idea is to remove layers of abstraction. This should also result in using less dependencies since those usually contribute to extra layers of abstractions. The lower level abstractions allow the application to have a lot more control over its foundation. This extra control should be used to make design decisions which could remove the need for added complexity in the long run. The less logic the application has to rely on to achieve the same features the easier it is to understand. The following sections explain the core ideas.

3.1 Design from the ground up

The first step for a project is to find out the baseline layer of abstraction which will be the lowest foundation for the project. Depending on the needed features find the lowest layer of abstraction which covers the needs. As Richard Feldman explained in his talk ”Hybrid-Level Programming” the layers of abstractions can limit an applications potential speed. In his talk he also explained that there really is no need to have so many layers of different abstractions and that it is often enough to just write a thin high level abstraction on top of the low level one which maximizes performance and control but does not compromise on usability for the users of the high level API [16].

The benefits of choosing the lowest possible layer might not only show in performance but also help keep complexity away. If there is less logic then there is less to learn and less possible spots of failure. For example communicating over the network it can sometimes be better to read the data straight from the socket, cast the raw bytes into a type and optionally do some integrity checks. In contrast to the traditional approach of using HTTP which in itself adds lots of complexity and some performance overhead due to the needed string parsing to extract the data. Also with HTTP the data is usually sent using JSON which also means having to have some parsing which ends up in performance overhead and again added complexity. In some cases it might be easier to just find out what is actually needed and create the simplest logic that handles it. Even relying on already existing libraries for HTTP it might not pay off in the long run due to the added complexity. Since in this case learning how to use the HTTP library correctly can be harder than learning the custom protocol which only does one simple thing. One could argue that a developer needs to learn HTTP only once but it is still mental overhead. The goal of this design is to keep mental overhead to a minimum.

Another way to view this concept is how Joel Spolsky said in his post about leaky abstractions: ”All non-trivial abstractions, to some degree, are leaky.”. He also explained that often abstractions do not really make our lives easier since they often leak and the underlying details still become relevant which means having to now know the inner workings and how those translate to the higher level logic [18]. The abstraction ”leaking” can be interpreted as the internal details starting to slowly become relevant and thus the abstraction starts losing its purpose of hiding information. The underlying logic that the application depends on should be minimal for just this reason, then there are less details that can leak. Having good abstractions is one thing, but if there are too many layers of abstractions and a large amount of information being hidden then it is much less likely for all the possible combinations of possible uses for these abstractions to not be leaky. By this definition the ideal case would be to start designing from the hardware, but the important part is to also factor in the actual need of control. Not all applications need to go this low level and a practical spot is to choose a fitting programming language and use the standard library it offers. But the lower the foundation is the more control there is for the design of the higher level abstraction which could serve as the new so called foundation for another layer.

In conclusion the principle of designing from the ground up means to identify the right abstraction from which to start from. It also includes finding the minimal logic needed to implement the features. There should also be some time dedicated to finding out if the features even need to be that complex and maybe changing the design can lead to reducing the needed logic. The goal should be to reduce mental overhead for the readers of the code and keep everything as simple as reasonably possible. The foundation should be used as a base on which future layers of abstraction can be built upon if the need arises. This whole ordeal can mean inventing something new and this behaviour should not be feared.

3.2 Designing abstractions

After the baseline abstraction is chosen then the application may suffer from having too much information exposed about the inner details. The natural step is to start creating functions and/or classes which encapsulate similar logic. In other words starting to create generalizing abstractions to gather similar types of functionality together and simplifying abstractions to hide the details which are not needed. The following is a list of rules for designing abstractions:

- Avoid leaky abstractions. An abstraction which leaks too often is not needed and can be seen as excess logic which does not help readability.

- Do not design abstractions from scratch, instead learn how others have done them and grab only the needed features. This can mean having to dive into the source code and/or internal documentation of open source projects.
- Prefer implementing abstractions from scratch. This means having full control over the implementation which means the abstractions can be made to perfectly fit into the project.
- Find the design that most probably will also work in the long run and implement only what is needed at the moment. This means having code that can be expanded later without large refactors but saves time when it turns out that the design needs changing.
- Since most time is spent reading code then optimize code for readability [17].
- Avoid creating abstractions which are used only once. It is hard to know what the abstraction should look like with only one use case.

These rules mostly apply to the logic which is the core functionality of the application. This is the logic which has the most effort put into it and where bad decisions can have the most negative impact. Usually a rule of thumb is to always ask what information is relevant for the current context of the code. Avoid hiding information which is actually relevant for the users of the abstraction. This means that some information is hard to hide since the problem just has that complexity in it. Trying to hide the complexity behind some abstractions only means that it will start to leak at some point. A better approach would be to try to redo the design which added the underlying complexity. If the design can't be optimized then the problem just has that complexity and the complexity should just be accepted. Overall the layers of abstraction should be kept thin and minimal. Every layer adds another mental overhead when there is a need to constantly jump between layers during development. The more the developers understand about the internals the more informed can be the decisions they make when writing new code.

3.3 Choosing dependencies

The following are rules to follow when deciding on if to use a dependency and what dependency to use:

- Consider using a dependency for logic which is not relevant to the core of the application. This means that the details of the implementation are not that relevant and thus it can be acceptable to use a dependency to cover that functionality.
- Use a dependency if the required logic is too complicated to write from scratch.

- Use a dependency if it is exactly what is needed and fits the goals of the application.
- If the dependency is on a protocol or concept (for example JWT, HTTP or communication with a database) then use a dependency which implements the needed specification. In these cases there is not that much that can be designed from scratch anyway since that might require changes to the specification.

The rules here are still quite vague and these should be thought of as possible options to evaluate but there certainly are edge cases and exceptions where these shouldn't be strictly followed.

License

Non-exclusive licence to reproduce the thesis and make the thesis public

I, **Raido Aunpuu**,

1. grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the digital archives of the University of Tartu until the expiry of the term of copyright, my thesis

   Analysing the Effects of Dependencies and Abstractions in Software Development,

   supervised by Dietmar Pfahl;

2. grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright;

3. am aware of the fact that the author retains the rights specified in points 1 and 2;

4. confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Raido Aunpuu
12/08/2025