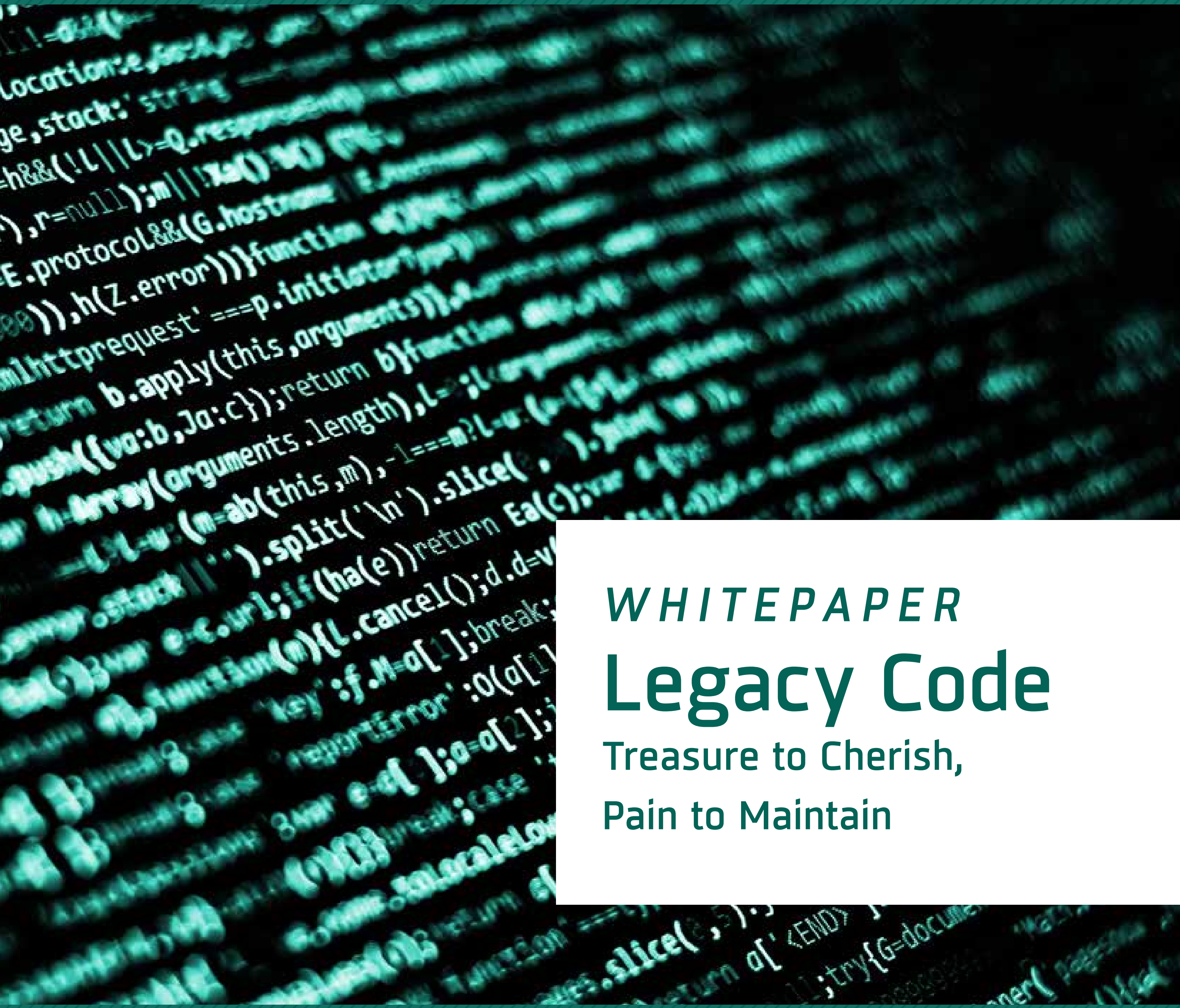
VORTECH



WHITEPAPER

Legacy Code

Treasure to Cherish,

Pain to Maintain

1. Introduction

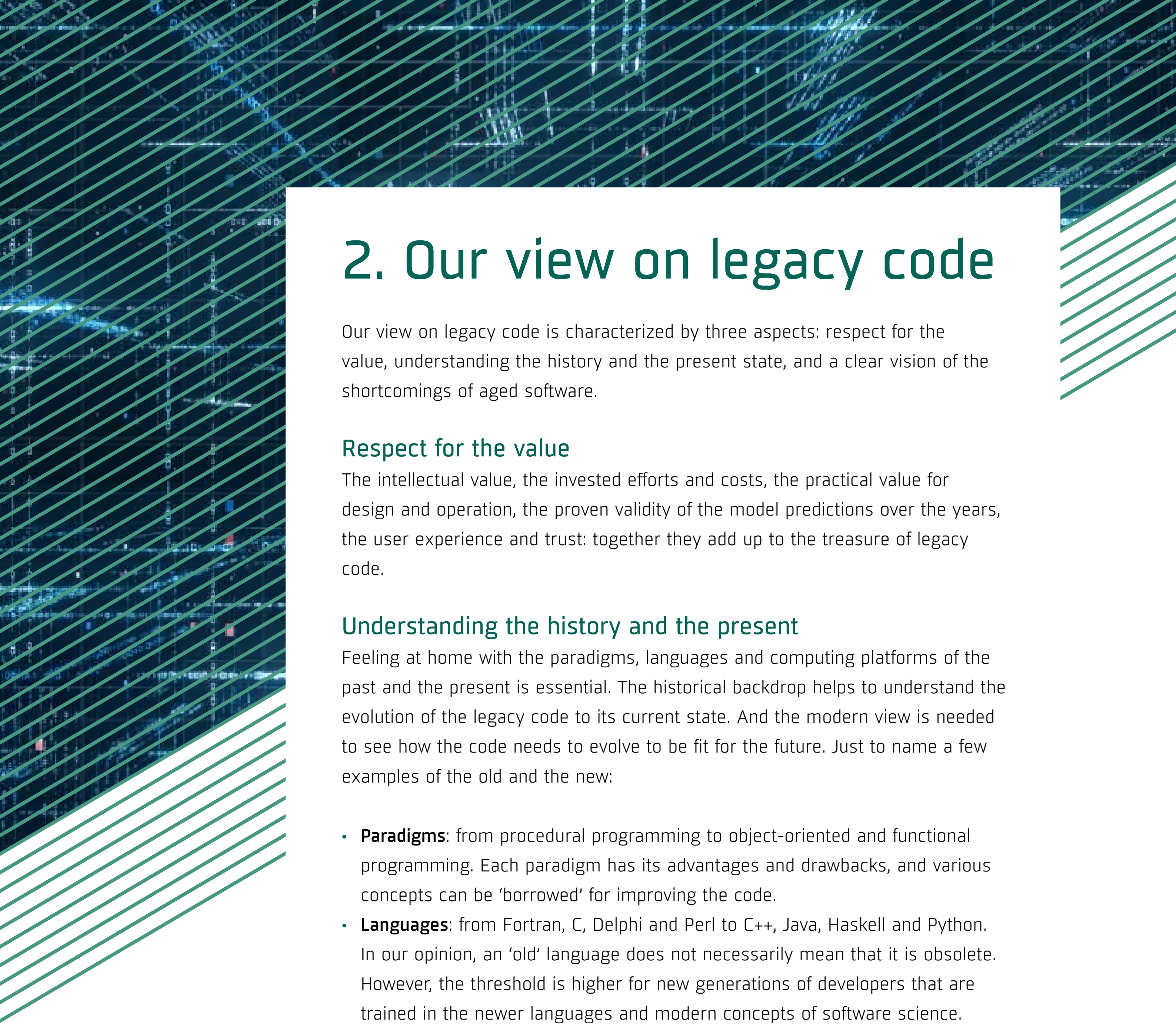
In our practice as scientific software engineers at VORtech, we work for organizations that have computational software as an essential part of their intellectual capital. Countless man-hours have been invested to get the software to its present maturity state. The wisdom, know-how and research of the experts in the organization's application domain are being captured in it and deployed through it. And as such it features as an indispensable tool in the design or operational stages of the organization's activities, or it is a key part of its product portfolio.

If successful, a computational software package can exist for many years and even decades. Over time, it evolves. New features and applications are being added, prediction results are being validated and improved, problems are being fixed. The experience with the software grows, as well as the trust in its usability and reliability. The software package easily outlives the hardware it is running on. And often it also outlives the developers and users within the organization. Thus, the package becomes a legacy treasure of great value.

At the same time however, the code base grows and becomes more and more difficult to manage. The efforts required to maintain and extend the software become extensive. This could go up even to the point that, despite the enormous investments, a from-scratch replacement seems the only way forward.

This whitepaper discusses how to deal with legacy code. How to keep it in a manageable state, and how to revive it if necessary. It is based on our daily practice with legacy code for more than 25 years, as well as the new insights and developments in software science that we have found useful. In the coming sections we describe

- our general view on legacy code,
- typical undesirable features of legacy code that we have come across,
- how these features have slipped into the code over time,
- what approaches can be taken to improve the code sustainability, and
- our general strategy to make the legacy treasure shine and prosper again.



2. Our view on legacy code

Our view on legacy code is characterized by three aspects: respect for the value, understanding the history and the present state, and a clear vision of the shortcomings of aged software.

Respect for the value

The intellectual value, the invested efforts and costs, the practical value for design and operation, the proven validity of the model predictions over the years, the user experience and trust: together they add up to the treasure of legacy code.

Understanding the history and the present

Feeling at home with the paradigms, languages and computing platforms of the past and the present is essential. The historical backdrop helps to understand the evolution of the legacy code to its current state. And the modern view is needed to see how the code needs to evolve to be fit for the future. Just to name a few examples of the old and the new:

- **Paradigms**: from procedural programming to object-oriented and functional programming. Each paradigm has its advantages and drawbacks, and various concepts can be ‘borrowed’ for improving the code.
- **Languages**: from Fortran, C, Delphi and Perl to C++, Java, Haskell and Python. In our opinion, an ‘old’ language does not necessarily mean that it is obsolete. However, the threshold is higher for new generations of developers that are trained in the newer languages and modern concepts of software science. Knowing both ‘old’ and ‘new’ lays the foundation for modernizing the legacy.
- **Platforms**:
  - From single-core CPUs to multi/many-core platforms with vectorization and pipelining optimizations, including GPUs and other accelerators.
  - From limited amounts of single-level memory that requires economic and explicit data management to hierarchical cached memory. Or alternatively, from explicitly managed memory allocations and cleanup to automatic memory management featuring garbage collection.
  - From cluster platforms requiring parallel computations featuring explicit data partitioning and data sharing protocols to cloud platforms running highly scalable and resilient stateless microservices.

Especially when the software is heavy on the computational load and when it is optimized for a specific platform, the migration to other platforms requires in-depth knowledge of both the ‘old’ and the ‘new’.

A clear vision of the shortcomings of aged software

Over time, the complexity of the software grows, and sub-optimal structures and patterns occur. The aging of software can make the code increasingly hard to understand, extend and maintain. Especially for new developers the learning curve becomes steeper and ridden with pitfalls. In the next section we list typical issues that are encountered with legacy code.

3. Typical aspects of legacy code

Various aspects of legacy code degrade the code quality and deteriorate the code readability. This makes continued maintenance and development increasingly difficult. The following aspects we have often seen to occur:

- **Entangled functionality**: procedures perform multiple tasks that are intertwined throughout the code.
- **Global variable scope and state**: any variable can be modified from everywhere throughout the code, and the state of the variable can affect every operation.
- **The Swiss army knife**: the application is used as a multi-purpose tool, as it is being employed for multiple use modes or workflows. Some of those use modes may not even be relevant anymore, but they are still present throughout the code. The program flow is controlled by global control switches throughout the code. Alternatively, various locations present the developer with an intimidating list of complicated conditional branches for distinguishing between the different workflows.
- **Insufficient testing facilities**, limited testing coverage of the various program states and use modes, and/or inadequate testing methods.
- **Low-quality technical documentation**, no clear explanation of the background of the algorithms in the code and the underlying design choices.
- **Self-made or obsolescent tooling** for managing the code, compiling, building, and deploying, for version control and issue tracking. Build environment requiring hidden environmental conditions or a complex manual process.

4. Causes of software aging

The typical downsides of legacy code are caused by two factors that are common in aging: evolution and decay.

Evolution

In principle, evolution of the software package is desirable because it expands the usefulness of the code and keeps it relevant for the current challenges. However, while the software evolves and matures, it is being deployed in ways that were not foreseen in its original design. The software outgrows its original application scope. This can cause a misfit between the original assumptions and constraints and today's requirements. Software structures that were conceived for the original situation may be unfit for incorporating the new situation. Examples of evolution are:

- Growth of complexity: new algorithms, increased dimensionality and problem size, more and more use cases.
- Use of the application not only as a design support tool but also in an operational setting as a digital twin.
- A changing base of users and developers, in other roles, new generations, featuring external parties.
- Employment of the application on other platforms such as Linux/Windows, in a web/cloud-based setting, on an embedded system.
- Moving from a stand-alone application to a multi-client service.

Without a proper refactoring effort, the new situation is then 'squeezed' into the old format which gives rise to an inadequate organization of data and code. Over time, a seemingly workable situation easily develops into a maintenance nightmare.

Decay

The deterioration of the code quality comes from a gradual buildup of inefficient code. This is also known as technical debt or postponed maintenance. Underlying causes of this deterioration in software quality are:

- The need for speed for critical operation or pressing timelines results in a quick fix or workaround that is never turned into a decently integrated solution.
- A lack of deep understanding of the code's purpose by the developer results in sub-optimal code. This often results in unnecessary code duplication, encapsulation, and layering.
- A limited budget for basic code maintenance which is often caused by a project-driven development approach.

5. Three approaches for dealing with legacy code

On a high level, three approaches can be selected to continue working with the legacy application: encapsulation, modernization, and full replacement.

Encapsulation

The legacy code kernel is treated as a black box, which is packed in a container and that is not to be opened anymore. A wrapper is created that takes care of the access to and from the kernel, preferably in a modern language such as Python.
Typically, this approach satisfies specific use cases such as running in parallel, interacting with other software or a new user interface.
An advantage of this approach is that the risk for introducing errors and bugs is relatively low. The kernel code is not touched, and the focus lies on the development of the new feature. This also makes the initial investment smaller than the approaches below.
On the other hand, in the end this approach can lead to a more complex (multi-application, cross-language) situation. And as the kernel remains untouched, there is no insight gained on its functionality. The legacy code is 'mummified' instead of 'revived', and in the long term it may result in an even more inflexible and unworkable situation.

Modernization

The code is gradually refactored to a manageable state and towards higher flexibility. Typical steps in this refactoring process include (not necessarily in this order):

- Gradual modularization by separating functionality into program units such as the user interface, file communication, computational kernel, administrative layer. Work towards the single responsibility principle, where each section of code (a procedure or function) performs one task alone.
- Extension of the test suite. The test suite contains test cases that execute the workflows, including all options that should be supported. It is the condensation of the desired functionality. As such it consolidates the application's behavior, and it guarantees that this behavior is conserved in future developments.
- Potentially: migration of (parts of) the code to a modern language. With a higher degree of modularity, this can be done in a partial/gradual approach, which makes it more manageable than a full migration of the entire application.
- Incorporation of new demands. With the modernization of the code through modularization, improved testing and migration, the current day needs, and

future perspectives can be met at a significantly lower cost. Examples of these demands include:

- new applications, models, and features,
- new interfaces or GUIs (for example web-based),
- thread-safety and concurrency,
- improved performance,
- running on new platforms, such as operation in the cloud or running on accelerators (GPUs).

The main advantages of this approach are predominantly its graduality and its aim for reviving the legacy code. It is relatively easy to divide the development into separate phases and to manage them in terms of time and result. Along the way, the insight grows about the purpose of the code, and thus the code documentation is gradually improved. The legacy is more and more unlocked, and its original luster is restored. Most often, hidden bugs that lurked in the code for a long time are uncovered and resolved. With the growing modularization, the new features and demands can be much better integrated into the code.

Downsides of the modernization approach are that the risk of introducing bugs and of managing the process are higher than with the encapsulation approach. Moreover, it feels like the 'hard' way to dive into the code to understand and refactor it. But this is often the only way. And the other approaches do not prevent the need for diving into the code, as both also require knowledge of what the code is supposed to be doing.

Full replacement

This is typically the last resort, and it comes with high cost and high risk. However, especially when the legacy code depends on obsolete platforms or technologies, the reincarnation approach is sometimes unavoidable. The desirable situation is that at the end of the replacement project, an application is realized that performs the same as the legacy application but that is ready for the future.

It is typically hard to sell to the management that the replacement, after considerable investment, will (hopefully!) provide the same capabilities as the original application. For applications that are still actively being developed this requires a period of double work, as the latest improvements also need to be incorporated in the replacement. On top of this, the risks of running into technical complications and consequent project delays are high. Apart from successful endeavors, we have also seen projects which dragged on and where the replacement software itself turned into legacy code, sometimes even before it was released.

6. How to relieve the pain in legacy code while keeping the treasure

To determine the best strategy for a specific application, several aspects must be clarified regarding the state of the code and its longer-term perspective, expectations and road map as laid out by the application owner. Whenever we embark on a new software modernization project, we typically begin with an assessment of the application and its perspectives. We perform an analysis of the state of the code and its technical surroundings. Moreover, we obtain clarity on the current and desired use, on today's requirements and tomorrow's perspective through discussions with the development team and the management. From this assessment, we draw up our recommendations for the best renovation strategy and we present an action plan. This analysis, which we refer to as the 'Model Scan', provides a long-term vision for the legacy application, and a guide for prioritizing and planning the application development. A Model Scan will range from as little as two days when the situation is clear up to ten days for more complex situations.

With a revived code base, the legacy has turned into an active project that is ready for the coming decades to support new applications, features and platforms. As an added benefit, the renovation process has increased your insight in the internal operation of the application, and this has inspired further advancements. And the cost of maintaining and expanding the software package has been considerably reduced. The legacy code has regained its luster and it has turned from an innovation blocker into an enabler.

VORTECH

Want to know more?

If you are dealing with a situation around legacy code, feel free to contact us. We can assist you with consultancy, project management and hands-on work. Contact us through info@vortech.nl or +31(0)15 285 01 25 for an appointment with one of our experts.

scientific software engineers