Summary: https://cr.yp.to/bib/1995/wirth.pdf

Wednesday, February 1, 2023 11:57 AM

The author argues that software has become too complex and bulky, and calls for a return to simplicity and elegance in software design. He notes that software should be written to serve human needs, rather than satisfying the desires of computer hardware or software vendors. He also emphasizes the importance of using clear, concise, and well-structured code, as well as focusing on ease of maintenance and modification.

Wirth suggests that software engineers should prioritize functionality and readability over optimization, and avoid adding unnecessary features or functionality. He also advocates for writing code in a way that is modular and easily maintainable, and for testing code thoroughly to catch bugs before release.

In conclusion, the author's plea is for a return to simplicity in software development, and for software engineers to focus on meeting the needs of users and producing code that is easy to understand, modify, and maintain.

POINTS:

- Complexity and bulkiness of software: The author argues that software has become overly complex and bulky, making it difficult to maintain and modify. He notes that this has resulted in a decline in reliability and an increase in the amount of time and resources required to develop software.
- Focusing on human needs: The author asserts that software should be designed to meet the needs of users, rather than satisfying the demands of computer hardware or software vendors. He notes that this requires a shift in focus from optimization to functionality and usability.
- Importance of clear and concise code: The author emphasizes the importance of writing code that is clear, concise, and well-structured. He notes that

this makes the code easier to understand, modify, and maintain.

- Modularity and maintainability: The author advocates for writing code that is modular and easily maintainable. He notes that this makes it easier to debug and update the code, and reduces the likelihood of introducing new bugs during modifications.
- Avoiding unnecessary features: The author suggests that software engineers should avoid adding unnecessary features or functionality to their code. He notes that this can make the code more difficult to understand and maintain, and can increase the likelihood of bugs and security vulnerabilities.
- Testing: The author notes the importance of thoroughly testing code before release to catch bugs and ensure that it works as intended. He also emphasizes the need for continuous testing and debugging throughout the development process.
- Hardware advancements: As computer hardware has become more powerful, software engineers have been able to add more features and functionality to their code. This has resulted in larger and more complex software programs.
- Marketing pressures: Software vendors may be pressured by market demands to add more features to their products in order to differentiate themselves from competitors. This can result in software that is bloated with unnecessary features.
- Lack of focus on usability: Software engineers may be more focused on optimizing their code for performance or for specific hardware configurations, rather than making it usable and accessible for users.
- Complex programming languages: The author argues that some programming languages are unnecessarily complex, which can make it more difficult for software engineers to write clear and concise code.
- Inadequate testing: Software engineers may not be dedicating enough time and resources to

thoroughly testing their code before release, leading to bugs and security vulnerabilities in the final product.

- Conclusion: In conclusion, the author's plea is for a return to simplicity in software development, with a focus on meeting the needs of users and producing code that is easy to understand, modify, and maintain. He argues that this will result in more reliable and usable software, and will reduce the amount of time and resources required to develop and maintain software.

Explain about Oberon in this paper

The author argues that the Oberon language is an example of a lean and elegant programming language that is well-suited for developing simple, clear, and maintainable software. He notes that the language has a simple syntax and minimalistic design, which makes it easier for software engineers to write clear and concise code.

Wirth also notes that the Oberon language is well-suited for developing modular and maintainable software, and that it has a number of features that make it easy to debug and update code.

In the paper, the author uses the example of the Oberon language to illustrate his argument that simplicity and elegance in software design can lead to more reliable and maintainable software. He also suggests that programming languages like Oberon could serve as a model for future programming language design.

Nine lessons learned from Oberon

In "A Plea for Lean Software," Niklaus Wirth outlines several lessons that software engineers can learn from the design of the Oberon programming language:

- Simple syntax: The Oberon language has a simple syntax that makes it easy to write clear and concise code.
- Minimalistic design: The Oberon language is designed with a minimalistic approach, which makes it easy to understand and use.

- Modular design: The Oberon language supports modular design, making it easy to write and maintain code.
- Support for debugging: The Oberon language has features that make it easier to debug and update code.
- Clear error messages: The Oberon language provides clear and concise error messages, which makes it easier to identify and fix bugs.
- Focus on functionality: The Oberon language is designed to support functionality over optimization, which results in more usable and accessible code.
- Well-documented: The Oberon language is well-documented, making it easier for software engineers to learn and use.
- Interoperability: The Oberon language is designed to be compatible with other programming languages and tools, making it easier to integrate with existing systems.
- Continual improvement: The Oberon language is designed to be continually improved and updated, making it more reliable and usable over time.

In conclusion, the design of the Oberon programming language can serve as an example for software engineers to follow in order to write clear, concise, and maintainable code that is well-suited for the needs of users.