University of Applied Sciences
HOCHSCHULE EMDEN·LEER

A Philosophy of Software Design

Introduction

Sangram Patil

7. Mai 2023

9 References

University of Applied Sciences
HOCHSCHULE EMDEN·LEER

About Author

About Author I

- John Ousterhout is Professor of Computer Science at Stanford University. His research addresses a wide range of topics related to infrastructure for building software systems, including distributed systems, operating systems, storage systems, development frameworks, and programming languages
- Prior to joining Stanford, John spent 14 years in industry where he founded two companies, preceded by another 14 years as a professor at Berkeley. Over the course of his career, Professor Ousterhout has built a number of influential systems (Sprite OS, Tcl.Tk, log structured file systems, Raft, RAMcloud, etc) and has taught several courses on software design.

About Author II

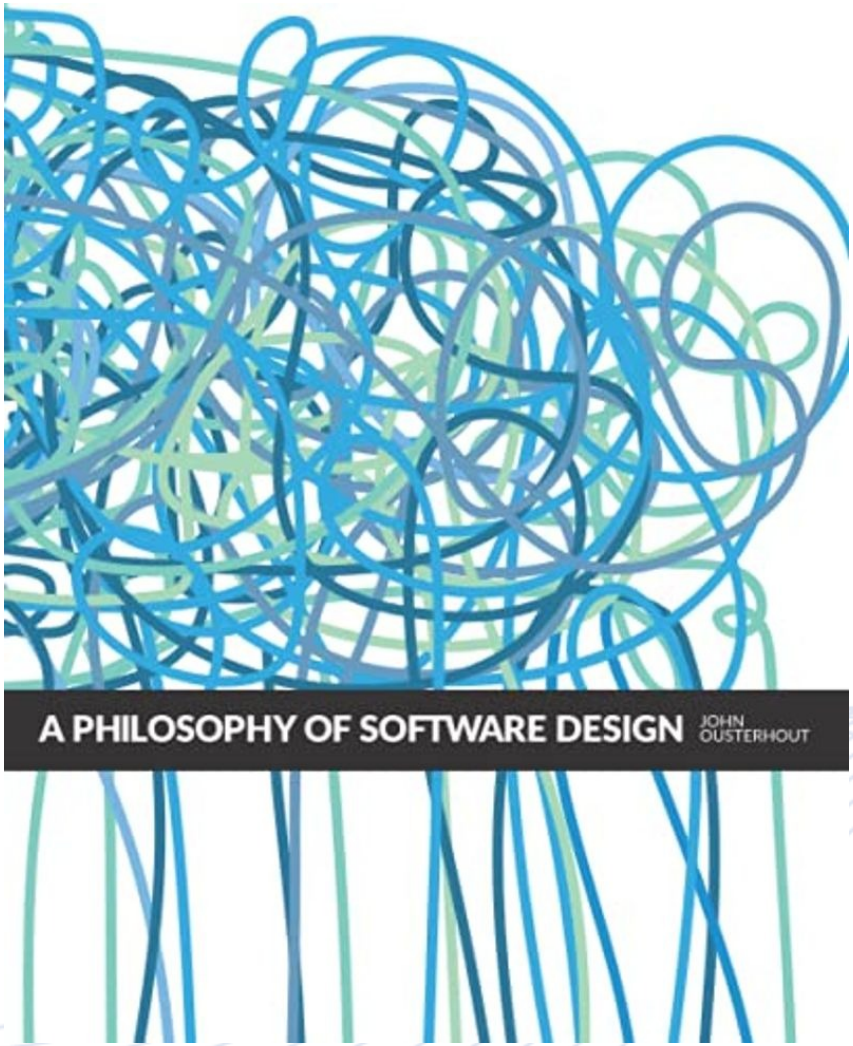

Abbildung: Book Cover

Programming as a Creative Activity

Programming as a Creative Activity I

- Writing computer software is one of the purest creative activities in the history of the human race
- Programmers have the ability to create virtual worlds with behaviors that do not exist in the real world, limited only by their imagination.
- Programming does not require physical skills like ballet or basketball, but rather a creative mind and the ability to organize thoughts.
- as with art, there is no single, correct way to program. It's down to the developer, their brain, and their imagination. The scope of what can be created with code is vast, and it's all down to the developer to design and architect.

Limitation of Understanding Systems

Limitation of Understanding Systems I

- Complexity accumulates as a program evolves and acquires more features, making it harder for programmers to keep all relevant factors in mind.
- Complexity slows down development, leads to bugs, and increases cost.
- Complexity increases inevitably over the life of any program
- Larger programs with more people working on them become even more difficult to manage due to increasing complexity.

Managing Complexity I

- Development tools can help manage complexity, but there is a limit to what they can do.
- Simplifying software design is crucial to make it easier and cheaper to build more powerful systems.
- Simpler designs allow for building larger and more powerful systems before complexity becomes overwhelming.

Approaches to Fighting Complexity in Software Development

Approaches to Finding Complexity in Software Development I

- Complexity in software can be addressed through two general approaches.
  1. Simplify Code for Reduced Complexity
  2. Encapsulate Complexity with Modular Design

1.Simplify Code for Reduced Complexity I

- Code can be made simpler and more obvious to reduce complexity.
- Eliminating special cases and using consistent identifiers can help simplify code.
- Simpler code is easier to understand and maintain, leading to reduced complexity.
- Simpler code involves writing code that is easy to read, understand, and maintain. Simpler code promotes efficient development, reduces bugs, and makes it easier to collaborate with other developers.
- For example, complexity can be reduced by eliminating special cases or using identifiers in a consistent fashion.

2.Encapsulate Complexity with Modular Design I

- Modular design is a second approach to addressing complexity in software.
- In modular design, a software system is divided into modules, such as classes in object-oriented languages.
- Modules are designed to be relatively independent of each other, encapsulating complexity.
- Programmers can work on one module without having to understand all the details of other modules.

Benefits of Modular Design I

- Modular design helps manage complexity by encapsulating it within individual modules.
- It allows for more focused and efficient development, as programmers can work on specific modules without getting overwhelmed by the entire system.
- Encapsulated complexity also promotes code reusability and maintainability.
- Modular design enables better team collaboration, as different team members can work on different modules concurrently.

The Waterfall Model

The Waterfall Model I

- The waterfall model is a traditional approach to software design where phases such as requirements definition, design, coding, testing, and maintenance are discrete and sequential.
- Design is concentrated at the beginning of the project, and the entire system is designed at once during the design phase.
- The design is frozen at the end of the design phase, and subsequent phases implement the frozen design.[PWB09]

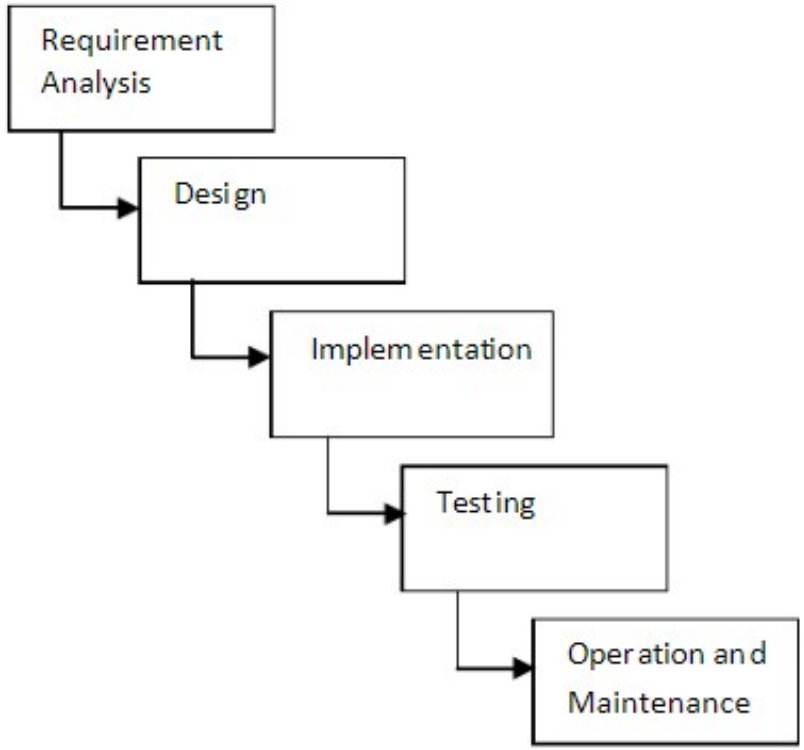

Abbildung: Block Diagram Of Waterfall

Challenges with the Waterfall Model I

- Software systems are inherently complex, and it's not possible to fully understand all the implications of the design before building the system.
- Problems in the initial design often become apparent during implementation, but the waterfall model is not structured to accommodate major design changes at that point.
- Developers may try to patch around problems without changing the overall design, resulting in an explosion of complexity.

Incremental Approach in Software Development

Incremental Approach in Software Development I

- Most software development projects today use an incremental approach. Initial design focuses on a small subset of overall functionality. Subset is designed, implemented, and evaluated.
- Problems with original design are discovered and corrected. Few more features are designed, implemented, and evaluated in iterations. Each iteration exposes problems with existing design, which are fixed before next set of features is designed.
- Spreading out design allows fixing problems with initial design while the system is still small. Later features benefit from experience gained during implementation of earlier features, resulting in fewer problems.

Benefits of Incremental Approach I

- Incremental development means software design is never done.
- Design happens continuously over the life of a system.
- Developers should always be thinking about design issues.
- Continuous redesign is a key aspect of incremental development. Initial design is rarely the best one; experience shows better ways to do things.

University of Applied Sciences
HOCHSCHULE EMDEN·LEER

How to use this book

Learning from Code Reviews and Design Alternatives I

- Many design principles discussed in this book may seem abstract without looking at actual code.
- The best way to use this book is in conjunction with code reviews
- When you read other people's code, think about whether it conforms to the concepts discussed in the book and how that relates to the complexity of the code.
- You can use the red flags described here to identify problems and suggest improvements. Reviewing code will also expose you to new design approaches and programming techniques.
- It's easier to see design problems in someone else's code than your own

Recognization of Red Flags I

- The best ways to improve your design skills is to learn to recognize red flags: signs that a piece of code is probably more complicated than it needs to be.
- In this book red flags are pointed out that suggest problems related to each major design issue; the most important ones are summarized at the back of the book.
- You can then use these when you are coding: when you see a red flag, stop and look for an alternate design that eliminates the problem
- When you first try this approach, you may have to try several design alternatives before you find one that eliminates the red flag
- The more alternatives you try before fixing the problem, the more you will learn. Over time, you will find that your code has fewer and fewer red flags, and your designs are cleaner and cleaner.
- Your experience will also show you other red flags that you can use to identify design problems

Moderation and Discretion in Applying Design Principles I

- When applying the ideas from this book, it's important to use moderation and discretion
- Every rule has exceptions and every principle has limits. Taking design ideas to extremes can lead to negative outcomes.
- Beautiful designs reflect a balance between competing ideas and approaches.
- In the book, Several chapters have sections titled "Taking it too far," which describe how to recognize when you are overdoing a good thing.
- Almost all of the examples in this book are in Java or C++, and much of the discussion is in terms of designing classes in an object-oriented language.
- Almost all of the ideas related to methods can also be applied to functions in a language without object-oriented features, such as C.

Conclusion

Conclusion I

- In conclusion, the book acknowledges that understanding design principles in theory may not be enough, and recommends using code reviews and practical examples to truly appreciate and apply the concepts discussed.
- Software design is a continuous process that spans the entire lifecycle of a software system.
- Red flags are highlighted as signs of potential design problems, and readers are encouraged to look for alternate designs to eliminate these red flags. However, it's important to exercise moderation and discretion, as taking any design idea to an extreme can lead to issues

References

References I

[PWB09] Kai Petersen, Claes Wohlin und Dejan Baca. “The waterfall model in large-scale development”. In: *Product-Focused Software Process Improvement: 10th International Conference, PROFES 2009, Oulu, Finland, June 15-17, 2009. Proceedings 10*. Springer. 2009, S. 386–400.

University of Applied Sciences
HOCHSCHULE EMDEN·LEER

Thank you

für Ihre Aufmerksamkeit