The End of Programming as We Know It

How Generative AI is Transforming Software Engineering

Jürgen Döllner

Hasso-Plattner-Institute for Digital Engineering

November 7, 2024

Introduction

- Generative AI is fundamentally changing software development and signaling the end of programming and software development as we know it.

Introduction

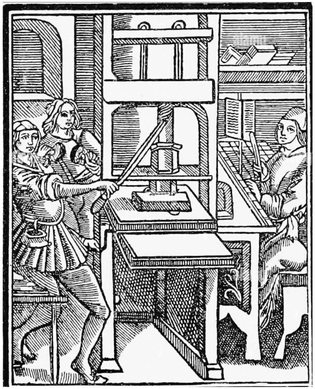


Figure: Democratization of knowledge: The Gutenberg printing press revolutionized access to information by enabling the mass production of books. Generative AI enables low-threshold code generation and therefore radically reopens software development to humans and machines.

The Landscape of Software Engineering

Practices

- Human-based analysis, design, implementation, and deployment tools and methods

Ingredients

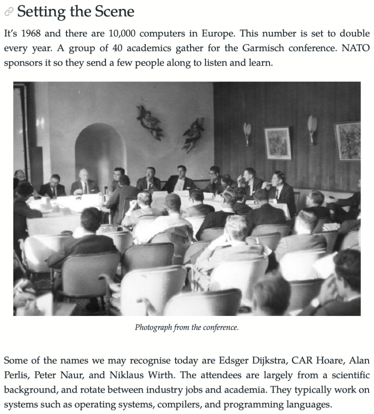
- Human logic and problem-solving, human problem understanding, predominant procedural thinking, thinking in human dimensions

Limitations

- Limited scalability, proliferation of artifacts, increasing complexity of legacy code, time-consuming and non-deterministic development processes (budget, time), ...

The Landscape of Software Engineering

Software Crisis



Setting the Scene

It's 1968 and there are 10,000 computers in Europe. This number is set to double every year. A group of 40 academics gather for the Garmisch conference. NATO sponsors it so they send a few people along to listen and learn.

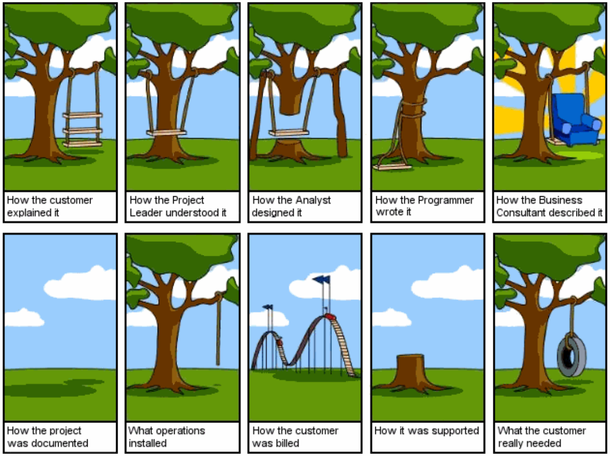
Photograph from the conference.

Some of the names we may recognise today are Edsger Dijkstra, CAR Hoare, Alan Perlis, Peter Naur, and Niklaus Wirth. The attendees are largely from a scientific background, and rotate between industry jobs and academia. They typically work on systems such as operating systems, compilers, and programming languages.

Figure: 1968: The NATO Software Engineering Conference coined the terms software *engineering* and *software crisis* recognizing the complexity and difficulty in software development (isthisit.nz).

The Traditional Landscape of Software Engineering

Software Crisis


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

The Rise of Generative AI in Software Development

Programming Languages are Languages

- Syntax and semantics are strictly defined
- Predictable patters are ideal for AI modeling
- Coding standards and conventions enhance learning

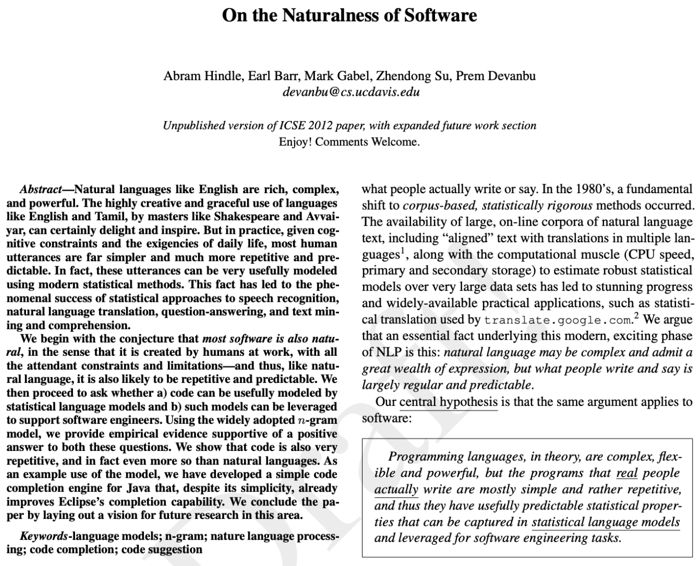
LLMs Trained on Code

- Exposure to vast codebases allows LLMs to learn and understand programmed artefacts
- Complementary information steams from topics over proramming (e.g., stackoverflow, ticket systems)

Synergy

- Structured data aligns with LLM sequence modeling
- AI models learn from both code and natural language, enhancing versatility

The Rise of Generative AI in Software Development



On the Naturalness of Software

Abram Hindle, Earl Barr, Mark Gabel, Zhendong Su, Prem Devanbu
devanbu@cs.ucdavis.edu

Unpublished version of ICSE 2012 paper, with expanded future work section
Enjoy! Comments Welcome.

***Abstract*—Natural languages like English are rich, complex, and powerful. The highly creative and graceful use of languages like English and Tamil, by masters like Shakespeare and Avvaiyar, can certainly delight and inspire. But in practice, given cognitive constraints and the exigencies of daily life, most human utterances are far simpler and much more repetitive and predictable. In fact, these utterances can be very usefully modeled using modern statistical methods. This fact has led to the phenomenal success of statistical approaches to speech recognition, natural language translation, question-answering, and text mining and comprehension.**

We begin with the conjecture that *most software is also natural*, in the sense that it is created by humans at work, with all the attendant constraints and limitations—and thus, like natural language, it is also likely to be repetitive and predictable. We then proceed to ask whether a) code can be usefully modeled by statistical language models and b) such models can be leveraged to support software engineers. Using the widely adopted n-gram model, we provide empirical evidence supportive of a positive answer to both these questions. We show that code is also very repetitive, and in fact even more so than natural languages. As an example use of the model, we have developed a simple code completion engine for Java that, despite its simplicity, already improves Eclipse's completion capability. We conclude the paper by laying out a vision for future research in this area.

***Keywords*-language models; n-gram; nature language processing; code completion; code suggestion**

what people actually write or say. In the 1980's, a fundamental shift to *corpus-based, statistically rigorous* methods occurred. The availability of large, on-line corpora of natural language text, including "aligned" text with translations in multiple languages[1], along with the computational muscle (CPU speed, primary and secondary storage) to estimate robust statistical models over very large data sets has led to stunning progress and widely-available practical applications, such as statistical translation used by `translate.google.com`.[2] We argue that an essential fact underlying this modern, exciting phase of NLP is this: *natural language may be complex and admit a great wealth of expression, but what people write and say is largely regular and predictable*.

Our central hypothesis is that the same argument applies to software:

> *Programming languages, in theory, are complex, flexible and powerful, but the programs that real people actually write are mostly simple and rather repetitive, and thus they have usefully predictable statistical properties that can be captured in statistical language models and leveraged for software engineering tasks.*

Transforming Programming Practices

From Coding to Solution Design

- Software developers spend a large part of time on design, coding, debugging, redesign, and testing code as well as on maintenance, refactoring, etc.
- Generative AI automates large parts of these activities
- Generative AI shifts focus on specifying what a solution should do, not how to implement it
- "Prompts", for example, are key ingredients of generative AI approaches

Transforming Programming Practices

Enhanced Human-Machine Collaboration

- Symbiosis between human insights and AI's speed and precision
- Generated code with high degree of quality, following best practices and reducing the likelihood of errors
- The time required for coding could probably be completely neglected in the future

Transforming Programming Practices

Software On-Demand

- If software can be automatically generated within a controlled environment for a given context, it can be generated on demand, i.e., just-in-time
- For example, the coding, testing, deployment and execution of software is only performed once, exactly when a service receives a specific request
- It will be no longer necessary to develop a general, all-encompassing software system in advance when a specific software system can be derived at any time
- Example: Exploratory data analysis by ChatGPT

Capabilities Beyond Human Programmers

Speed and Efficiency

- Rapid code generation and automated testing

Handling Complexity

- Managing large-scale frameworks and intricate algorithms

Continuous Learning

- AI models improve over time with more data

Genetic Programming

- AI models can generated an endless variety of codings to find an optimal solution

Consequences

Redefining Roles in Software Development

- Software engineers as strategists and supervisors
- Correctness and reliability checks as central tasks

Decline of Manual Coding

- AI taking over coding – from code pilots to fully automatic code generation
- Changes to the tool sets of developers

Evolving Skill Sets

- Need for AI oversight and system design expertise
- Ethics becomes a concern, avoiding biases and maintaining code integrity

Consequences

AI-Adapted Programming Techniques and Tools

- High-level programming languages (e.g., Python, Java, C++) are created with syntax and semantics that are easier for humans to read, write, and understand
- Generative AI could leverage patterns and structures that are efficient for AI algorithms but not necessarily intuitive for humans, i.e., optimize for *computational efficiency* rather than for human comprehension
- Generative AI will potentially generate *optimized machine code* directly, bypassing the need for intermediate high-level languages
- Generative AI could use *self-modifying code*, i.e., code that can alter its own instructions during execution to adapt to changing conditions or optimize performance
- Generative AI could use *non-instructional code representations*, e.g., graphs, or stateless approaches (e.g., functional programming).

Consequences

Example: Ultimate RISC

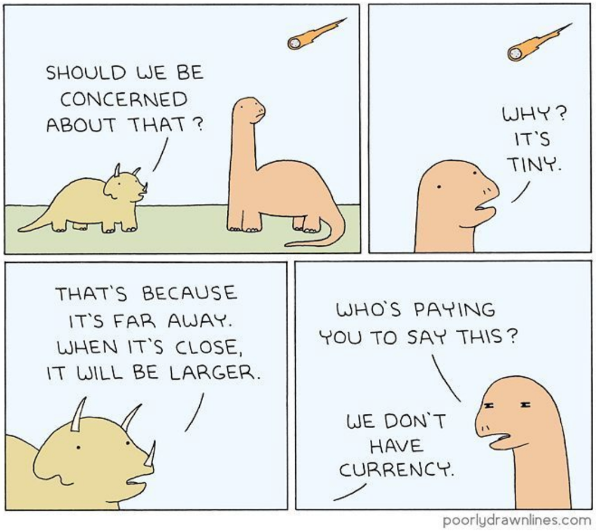
- What is the minimum number of instructions a CPU needs in order to be universal? – A single instruction.
- Example: "Subtract, Test, and Jump" (STJ) RISC
- `STJ(x,y,z)` $\Leftrightarrow$ `x := x - y; if x <= 0 then goto z;`

Consequences

System Architecture and System Modularization

- High-level system architecture and interfaces (e.g., APIs) will be subject of explicit and strategic human design
- One kind of module, modules below a certain complexity threshold, can be generated automatically based on high-level specifications and corresponding test data
- Another kind of module, having a clearly defined functionality, embeds an LLM (or an MLLM), that is, these modules will become *trained black-boxes*

Attention!


SHOULD WE BE CONCERNED ABOUT THAT?
WHY? IT'S TINY.
THAT'S BECAUSE IT'S FAR AWAY. WHEN IT'S CLOSE, IT WILL BE LARGER.
WHO'S PAYING YOU TO SAY THIS?
WE DON'T HAVE CURRENCY.
poorlydrawnlines.com

Last Slide

Thank You for Your Attention.

- **Contact Information:**
  - Email: doellner@hpi.de
  - Website: `https://www.hpi.de/doellner`