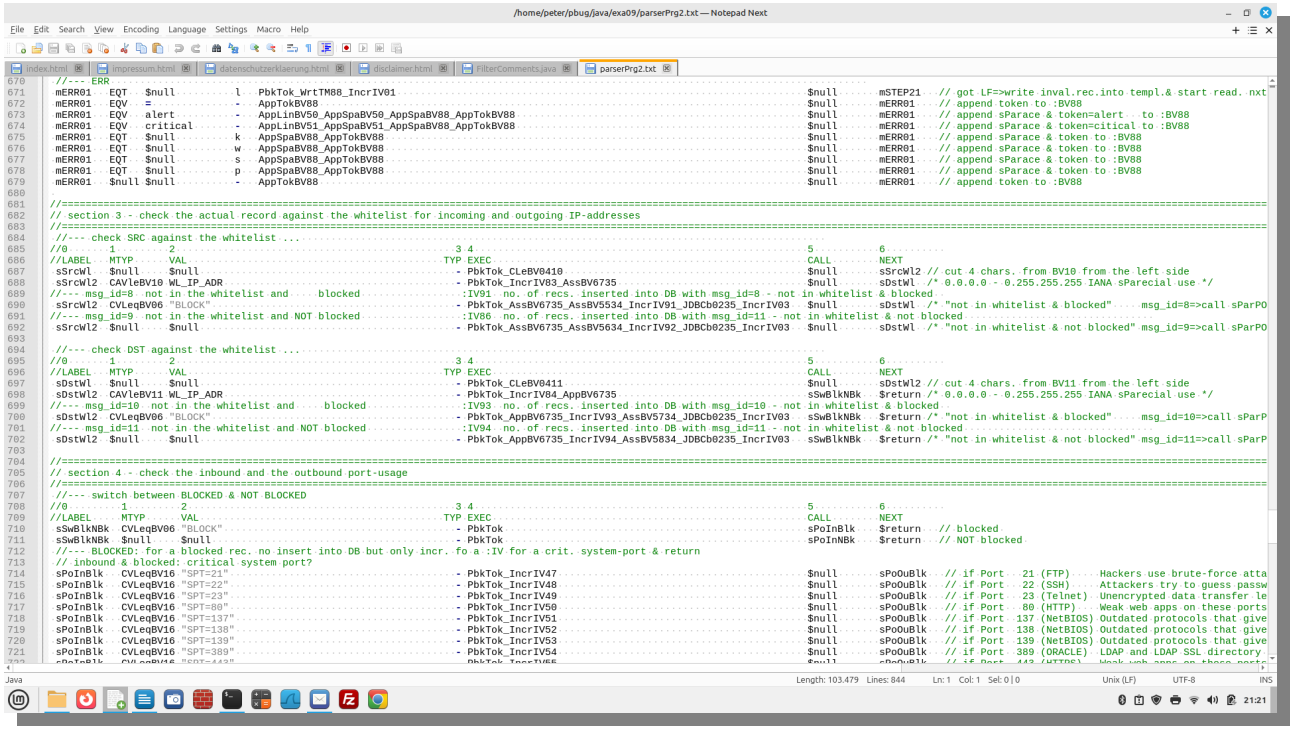
Table of Contents

Software development using AI....2
AI Training and Intellectual Property Issues....2
Format of the Generated Code....2
Quality of the generated code....3
Maintainability of Generated Code....3
A simple example of software architecture design....3

Software development using AI



AI Training and Intellectual Property Issues

Generative AI tools independently create code based on patterns derived from large datasets. To achieve this, generative AI models are trained using both publicly available and proprietary code. There is a risk that code of unknown origin may be used, potentially infringing upon ownership rights. Adherence to intellectual property rights and licensing terms is a critical consideration. Many generative AI tools reserve the right to use data provided by users for training purposes. There is a risk that code subject to an open-source license could be made publicly available. This could lead to violations of legal requirements and internal corporate policies. Consequently, oversight and analysis are necessary to review the generated code.

Format of the Generated Code

The format of the generated code is determined primarily by the training data rather than by the style guides of the company using the generative AI tools. Coding style guides provide specifications for:

- Naming conventions for custom identifiers—including permitted languages, prefixes and suffixes, singular/plural forms, capitalization rules, and readability standards,
- Capitalization of keywords,
- Variable scope (global vs. local),

- Formatting, indentation, and the use of tabs (to ensure a consistent appearance), spacing between operators and variables, and line lengths,
- Modularization to adhere to complexity metrics; and
- Guidelines for code comments, which can also be used to automatically generate technical documentation.

Adapting generated code to meet such specifications is highly time-consuming and can lead to errors.

Quality of the generated code

It may not be clearly evident whether the resulting software architecture is proven, robust, and performant. Code produced by AI tools might not always meet a company's quality standards. Generative AI tools rely on patterns and approaches derived from their training data. If the quality of this training data has not been verified, it can result in poor-quality or inefficient code. Substandard or inefficient code can lead to errors, security vulnerabilities, and a flawed software architecture. If generated code contains a specific security vulnerability, it may be possible to trace it back to the source material used to train the generative AI tool. In such cases, in addition to the damage caused by the security vulnerability, the organization could also face legal action for infringing upon the intellectual property rights associated with that code.

Maintainability of Generated Code

The maintainability of generated code can be complex and challenging. Even when software functions according to its specifications, the underlying logic may be difficult to follow or understand. A lack of transparency can render the generated code unmaintainable. Consequently, refactoring or debugging becomes impossible should errors arise. Attempting to compensate for a lack of experience among software developers by relying on AI creates significant risks for companies. Adhering to "Clean Code" principles fosters employee satisfaction and helps maintain low staff turnover rates within the IT sector. The loss of experienced personnel often entails substantial costs and risks for a company.

A simple example of software architecture design

As already described above is Software development using AI, in essence, software development using code generators, an approach that carries significant risks. There is a danger that the system may generate code fragments that are not fully understood, making it impossible to verify the correctness of the resulting code. Furthermore, there is a risk that the generated software architecture may become too complex to maintain. Such code is also referred to by software developers as "spaghetti code." With this article, I would like to offer advice on how to avoid spaghetti code.

Software architectures vary according to their application domain:

- **Real-time** such as operational dashboards & IoT, messaging & chat platforms like WhatsApp, or financial trading platforms like Robinhood.

- **Batch Processing** like payroll processing, bank reconciliation & end-of-day settlements, utility & credit card billing or ETL (Extract, Transform, Load) & data warehousing.

The following describes an approach for batch processing. Using the "Jackson Structured Programming" methodology, it is possible to identify the individual components that will make up the system to be designed. (A detailed article on this topic can be found on Wikipedia.) Each component can often be viewed as a work step, encompassing several technical functions designed to perform one specific technical task each.

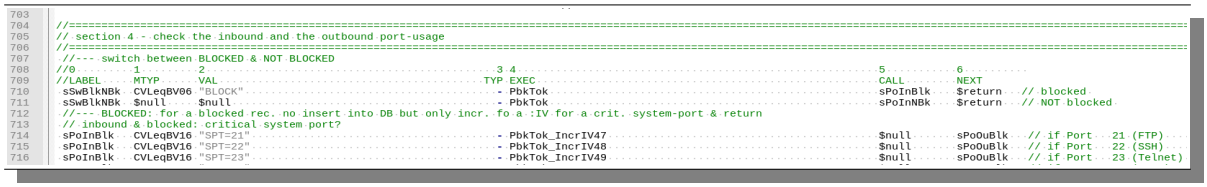
```
//-------------------------------------------------------------------------------------------------
// returning a write-command for the act. and the prev.char
//-------------------------------------------------------------------------------------------------
final private static int automat22() {
    /*
    |  /  |  *  |  -  |  #  |  lf |  '  |  "  |  so |
    +-----+-----+-----+-----+-----+-----+-----+-----+
    1 |  2n |  1a |  5n |  6n |  1a |  7n |  8n |  1a | / - # ' " (unmaskiert) Statusänderung
    +-----+-----+-----+-----+-----+-----+-----+-----+
    2 |6n/2v|  3n |  5v |  6v |  1z |  7v |  8v |  1z | / * - # ' " (unmaskiert) Statusänderung - // Einzeilenkommentar=6n c
    +-----+-----+-----+-----+-----+-----+-----+-----+
    3 |  3n |  4n |  3n |  3n |  3n |  3n |  3n |  3n | * Statusänderung - Mehrzeilenkommentar ein
    +-----+-----+-----+-----+-----+-----+-----+-----+
    4 |  1n |  4n |  3n |  3n |  3n |  3n |  3n |  3n | / Statusänderung - Mehrzeilenkommentar aus
    +-----+-----+-----+-----+-----+-----+-----+-----+
    5 |  2v |  1z |  6n |  6v |  1z |  7v |  8v |  1z | / - # ' " (unmaskiert) Statusänderung - Einzeilenkommentar mit Start
    +-----+-----+-----+-----+-----+-----+-----+-----+
    6 |  6n |  6n |  6n |  6n |  1n |  6n |  6n |  6n | lf Statusänderung - Einzeilenkommentar aus ?
    +-----+-----+-----+-----+-----+-----+-----+-----+
    7 | 7n/a| 7n/a| 7n/a| 7n/a| 7n/a| 7n/a| 7n/a| 7n/a| ' aus ?
    +-----+-----+-----+-----+-----+-----+-----+-----+----------------------------------------
    8 | 8n/a| 8n/a| 8n/a| 8n/a| 8n/a| 8n/a| 8n/a| 8n/a| " aus ?                 filDouSla   // double slash
    +-----+-----+-----+-----+-----+-----+-----+-----+----------------------------------------
    |  /  |  *  |  -  |  #  |  lf |  '  |  "  |  so |                         filDouHyp   -- double hyphen
    +-----+-----+-----+-----+-----+-----+-----+-----+----------------------------------------
    n nichts schreiben                                                        filNumSgn   #  number sign
    v vorheriges Zeichen schreiben                                            filSinQuo   '  single quotation
    a aktuelles  Zeichen schreiben                                            filDouQuo   "  double quotation
    z vorheriges und aktuelles Zeichen schreiben                              mask_q    b/d mask single quote by backslash
                                                                              mask_Q    b/d double-quote by doubling
    */
```

A finite-state machine (or simply a "state machine") is well-suited for this purpose. (Wikipedia also provides an in-depth article on this subject.) By saving the current state to a hard drive or database, the system can be restarted following a failure, allowing it to resume processing from the exact point where it left off once the error has been resolved. This capability is highly recommended. This approach results in a robust software architecture. Once established, you can have an AI agent generate the individual technical functions—unless, of course, you already have proven, reliable modules in your own personal software library.

It is advisable to design a state machine for each processing layer. Each layer should be assigned a unique identifier (or abbreviation) that is included in every log entry and error message. Every message should be assigned a message number that is unique across the entire system. This makes it possible to pinpoint the exact location in the code that triggered a given message; furthermore, if an error leads to the program terminating, one can also view the call stack.

The text analysis framework presented here is structured as follows:
1. Filtering of comments
2. Identification of tokens
3. Identification of keywords
4. Parsing of tokens using a scripting language

Each layer includes a program controlled by a state machine. The parser operates using a state machine that incorporates a stack, allowing for the parsing of nested SQL statements.



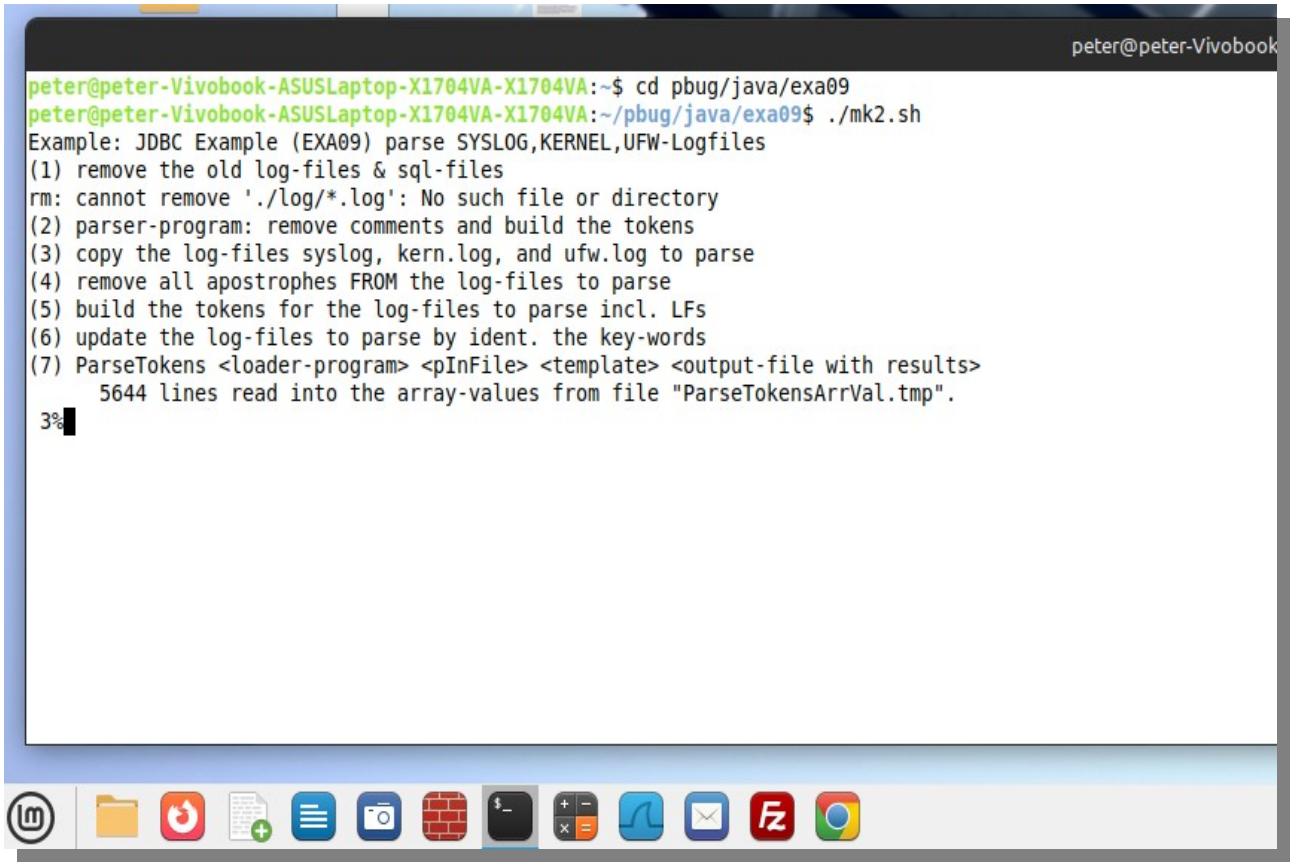
```
//=============================================================================================================================================================
// section 4 - check the inbound and the outbound port-usage
//=============================================================================================================================================================
//--- switch between BLOCKED & NOT BLOCKED
//0        1        2                                                3 4                                                  5         6
//LABEL    MTYP     VAL                                            TYP EXEC                                             CALL      NEXT
 sSwBlkNBk CVLeqBV06 "BLOCK"                                        - PbkTok                                             sPoInBlk  $return   // blocked
 sSwBlkNBk $null     $null                                          - PbkTok                                             sPoInNBk  $return   // NOT blocked
//--- BLOCKED: for a blocked rec. no insert into DB but only incr. fo a :IV for a crit. system-port & return
// inbound & blocked: critical system port?
 sPoInBlk  CVLeqBV16 "SPT=21"                                       - PbkTok_IncrIV47                                    $null     sPoOuBlk  // if Port 21 (FTP)
 sPoInBlk  CVLeqBV16 "SPT=22"                                       - PbkTok_IncrIV48                                    $null     sPoOuBlk  // if Port 22 (SSH)
 sPoInBlk  CVLeqBV16 "SPT=23"                                       - PbkTok_IncrIV49                                    $null     sPoOuBlk  // if Port 23 (Telnet)
```

If one defines the necessary components for a 3GL (third-generation language) as including

- expressions (or sequences),
- conditions,
- loops, and
- subprogram calls,

then the parser's scripting language can be classified as a simple 3GL.
This language makes it possible to design custom-tailored parsers.

Conclusion: By designing a suitable software architecture and limiting the use of AI to the generation of code for precisely defined technical functions, the risks associated with employing this technology should remain manageable.



An Example:

A finite state machine with a stack, also known as a Pushdown Automaton (PDA / in German Kellerautomat) in theory, stores the call-stack.

Ein endlicher Automat mit einem Stack – in der Theorie auch als Kellerautomat (englisch: Pushdown Automaton, PDA) bezeichnet – speichert den Aufruf-Stack (Call-Stack).

Ein Beispiel für einen Kellerautomaten findet sich im TAFr-Handbuch, im Kapitel „The Parser Program“ unter „Introduction“:

- **An event is a command from the work-list, which is the call of an app or a working-step (a function in a 3-GL app):**

event / state	event-1	event-2	event-3	event-4
1	START	START		
2	STP00			
3	STP01	STP01	STP01	
4	STP02	STP02	STP02	
5	STP03	STP03	STP03	
. . . . .				

- **A call (green) jumps to another position in the actual or another transition table. To jump back to the calling position a RETURN-command is to be processed:**

state \ event	event-1	event-2	event-3	event-4
1				
2	STP02	STP02 ERR01	STP02 ERR01	
3				
4				
5				

```
Layers           Technical Functions
Pre-processing  : check of the work list
                  check of the hard drive storage space
                  check of the condition of the database
                  etc.
Input           : read the next task from the work list
Process         : process the actual task
Output          : write the output into the database
                  write the data into the store on the hard disk
                  etc.
Post-processing : Write log-messages into the log-file,
                  process clean up (delete temp. file)
                  etc.
```

I leave the final design to the interested reader.