SourceMonitor Next Generation (SM-NG)

Jim Wanner retired from software development. The software SourceMonitor will become open source. There are several things from preventing the code completely accessible for everyone e. g. licensed code, generated code of abandoned tools and other challenges.

About SourceMonitor

The programm is able to collect metrics from source code:

- Collects metrics in a fast, single pass through source files.
- Measures metrics for source code written in C++, C, C#, VB.NET, Java, Delphi, Visual Basic (VB6) or HTML.
- Includes method and function level metrics for C++, C, C#, VB.NET, Java, and Delphi.
- Offers Modified Complexity metric option.
- Saves metrics in checkpoints for comparison during software development projects.
- Displays and prints metrics in tables and charts, including Kiviat diagrams.
- Operates within a standard Windows GUI or inside your scripts using XML command files.
- Exports metrics to XML or CSV (comma-separated-value) files for further processing with other tools.

Motivation

In my life as a software developer I have come to appreciate and love SourceMonitor. After doing an update, I read that Jim will no longer be actively developing SourceMonitor. When I suggested him to make SourceMonitor OpenSource, we got more in contact. I would like to make this great project OpenSource and develop it further.

Decissions made

- Continue the SourceMonitor project in the sense of Jim Wanner
- Work on SourceMonitor to make it open source
- Programming language: C++
- License: MIT

Used tools and frameworks

- **20221029** - Use CMake in minimum version 3.5
- **20221029** - Use Qt for UI în version 6.2 LTS

Used tools

SourceMonitor itself

SourceMonitor uses various technologies and tools. These include

- C++ as programming language
- MFC for the user interface
- Inno Setup for the installer

- Licensed code for graphics
- Several language parsers, based on licensed and abandoned software

This Documentation

For this documentaiton some public available tools are used. There are

- Visual Studio Code as editor
- Draw.io Integration as plugin for the graphics
- Markdown PDF as plugion to convert markdown to PDF
- markdwonling as plugin for correct Markdown

Inventory

The latest version of SourceMonitor contains

- 230 files
- 67.150 Lines of code
- 39.009 Statements
- 195 Classes

This information is retrieved using SourceMonitor 3.5.16.49 and contains

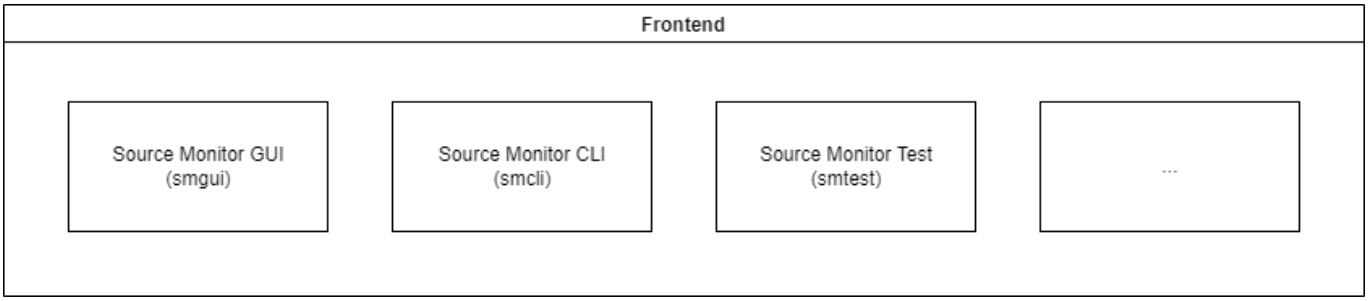
- SourceMonitor code
- Parser code: Generated by licensed and abandoned software
- Diagram code: External licensed library

Basic minds about future architecture

Shinichiro-san dropped in some minds about the future architecture. This is completely independend of the way how the project continues (either refacturing or start from scratch).

I've picked up these minds and continued with the work on it. There is not much success on it - there are several points I'm struggling with.

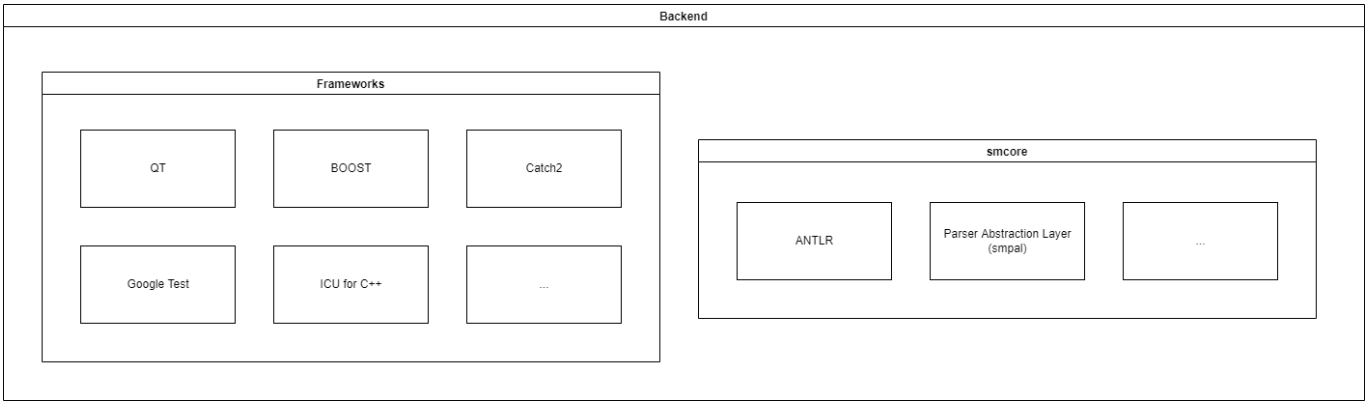
For the beginning, let's start with the draft design. There are several frontend modules which are using somehow functionality from the core. Frontend is defined as what is accessible to the user.



There is

- The GUI - the default frontend
- The CLI - the command line frontend
- The Test - the frontend for the internal tests of SourceMonitor

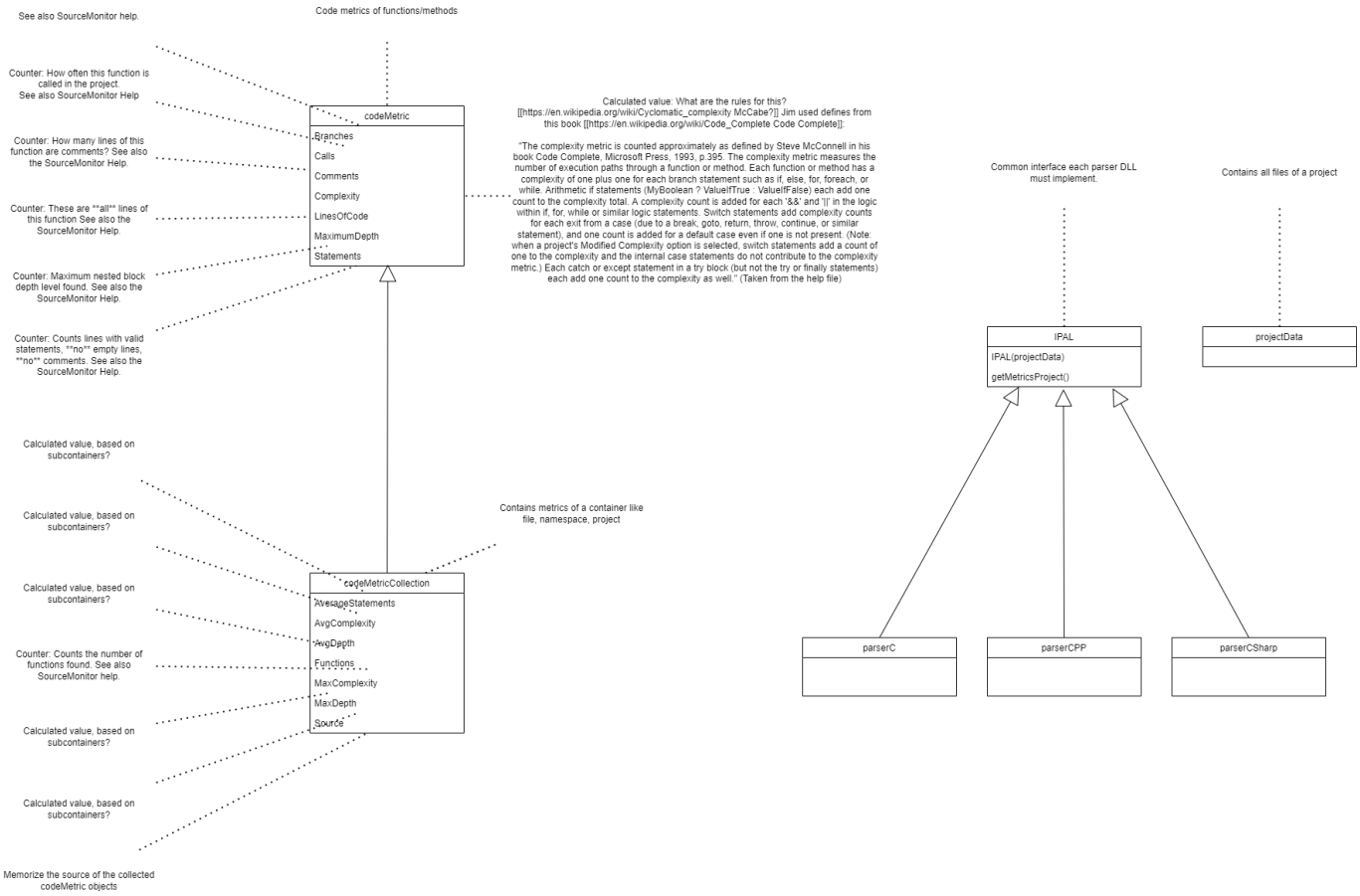
Based on the definition of the frontend, there is also a backend. This is defined as what is not accessible to the user and what is called from the frontend.



The backend also splits up into several parts as there are

- The used frameworks
  - boost
  - Catch2
  - Google Test
  - ICU
  - Qt
- The core functionality of SourceMonitor
  - Antlr4 or any other parser generator
  - The other core functionality like
    - Project handling
    - Calculation of the metrics based on parser information
    - Logging with plog or any other logging framework
    - ...

A draft design of the `smcore` called package will be



Considerations

What's the goal of SourceMonitor? To get information (metrics) about a project. In concrete about the code. So let's break a project down to the code. A project consists of several files. Each file can have classes, functions and namespaces. A class consists of properties and methods. The properties are not of interest for the metrics. The methods are defined by their signature and code. Functions are defined by their signature and code, too. And namespaces are containers for classes and/or functions.

- Project
  - Files[]
- File
  - Classes[]
  - Functions[]
  - Namespaces[]
- Class
  - Properties[]
  - Methods[]
- Method
  - Code
- Function
  - Code
- Namespaces
  - Classes[]
  - Functions[]
  - Variables[]

On the basis of the improvised architecture there are two kind of metric objects.

- codeMetric
- codeMetricCollection

Where the `codeMetric` object is used for functions and methods. All other containers like classes, files, namespaces and the project will be a `codeMetricCollection` and a container for the identified functions and methods. This considerations are based on regarding C++ - for other programming languages this might be different.

Processing

The processing of a project is described as followed:

- Collect all files of a project
- For each file
  - Abort if no parser is available (identified by the file extension?)
  - Parse file
  - Build up codeMetricCollection nodes and codeMetric during parsing

Challenges

There are many challenges. I've divided them up into technical challenges and software challenges.

Technical challenges

- CMake - there is a need for CMake knowledge to configure the whole project inclusive the used framework/libraries like the Antlr4 runtime
- Antlr4 - knowledge about this tool, too. The already provided grammars may need adjustments.
- `Refactoring` vs `start from scratch`
- (Unit-)tests for the code - there is a need of knowledge for test frameworks and writing tests.
- CI/CD - there is a need of knowing GitHub CI/CD.
- Multiplattform installer required.
- Translations for
  - Error messages
  - Logging entries
  - UI
- Creation of a new SourceMonitor home at SourceMonitor.github.io - knowledge about web development and GitHub CI/CD features.

Software challenges

Assuming that each language parser will be encapsulated in an own library, the data exchange between smcore and the parsing library might be difficult. Thinking about independent formats like

- Swap binary objects in memory
  - Pro: Fastest solution
  - Con: Could be a challenge regarding multiplattform goal.
- File based data exchange
  - Pro: Reader/Writer class can be used from both, parser and smcore

    - Con: This is a strong performance issue
- JSON based data exchange
    - Pro: JSON libs are available for C++, a benchmark comparison of various JSON genrators/parsers is listed here
    - Con: The used memory size might be problematic
    - Con: UTF-8 might be a problem
    - Con: The performance might be a problem

Open points

There are some points left where the SourceMonitor team needs to make some decisions. For some of them there are discussions already stareted.

- C++ level - C++11, C++14, C++17, C++20 or even C++23/26 is discussed here
- C++ frameworks are discussed here
- Development - `refactoring` vs `start from scratch` is discussed here
- Internationalization - ICU?
- Logging framework - plog?
- Parser - Antlr4?
- Plugin system for parser libraries?
- Software architecture?
- Used metrics of SourceMonitor are discussed here
- How to document
    - Code - doxygen?
    - Architecture graphics - drawio?
    - Help system - anyone of these?
    - Documentation - markdown?