**João Pedro Valadares Barrulas**

Bachelor in Computer Science

# Analysis of Code Blocks for Concern Detection in MATLAB Systems

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
**Computer Science and Informatics Engineering**

Adviser: Miguel Pessoa Monteiro, Assistant Professor, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Co-adviser: Nuno Miguel Cavalheiro Marques, Assistant Professor, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Examination Committee

| | |
|---|---|
| Chairperson: | Professor Artur Miguel Andrade Vieira Dias, PhD |
| Members: | Investigator João Carlos Viegas Martins Bispo, PhD |
| | Professor Miguel Jorge Tavares Pessoa Monteiro, PhD |

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

**December, 2019**

**Analysis of Code Blocks for Concern Detection in MATLAB Systems**

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

*To my family and pumpkin*

# Acknowledgements

Ao longo da elaboração desta dissertação, recebi um apoio e assistência enormes de diversas pessoas a quem quero agradecer. Em primeiro lugar, gostaria de agradecer ao meu orientador e coorientador, o Dr. M. Monteiro e o Dr. N. Marques, cujas experiências foram inestimáveies na formulação do tópico e metodologias a considerar. Agradeço a vossa paciência face às minhas dificuldades e perguntas e também por sempre me terem oferecido uma perspectiva positiva.

Gostaria de agradecer aos meus colegas, em particular o Bruno e o António, pela colaboração, pelos conselhos e ouvidos críticos, apoiaram-me e sempre estiveram dispostos ajudar. Também aos colegas que conheci numa visita à FEUP por terem contribuído para a escolha da direção certa.

Aos grupos de anhados (DQHSLN, Habitus est Omnia, anTUNiA) com quem muito cresci, só mostro arrependimento por não vos ter conhecido mais cedo. Obrigado por terem contribuído com um pouco de vós para a pessoa que está neste momento a terminar uma fase crítica e decisiva da sua vida. Na ausência dos nossos momentos de existência e partilha sei que não estaria aqui assim.

Finalmente, à minha família e cara metade, que me deram todo o apoio na deliberação sobre o que era certo para mim, além de me darem toda a motivação, comida caseira e recursos financeiros para descansar o meu sentido fora dos meus estudos. Nunca vos poderei agradecer o suficiente por todo o apoio que me deram ao longo da vida e de todos estes anos de faculdade.

# Abstract

It is known that the support provided by MATLAB for module decomposition is limited. Such limitations give rise to code symptoms, which can be explored for the development of techniques for the detection of concerns, namely unmodularised concerns. Recent work in the area of concern detection in MATLAB systems identified several recurring code patterns that can be associated to the presence of specific concerns. Some of the concerns detected proved to be unmodularised: they cut across the MATLAB system's modular decomposition.

The techniques already developed for detecting unmodularised concerns in MATLAB systems still lack precision and accuracy. As proposed in previous work, the techniques and tools for pinpointing and representing concern-detection patterns need maturing.

This thesis contributes with a more accurate structure for representing MATLAB code bases in an intelligent repository for MATLAB code, developed prior to this work. It perfects the structure representing MATLAB code on which the repository is based, by refining the notion of code block, and collects code patterns found in previous publications aggregating them into a catalogue. Subsequently, a preliminary study is made on the application of codes of blocks for the detection of concerns, validating previous concern related patterns and evaluate the existence of new ones.

**Keywords:** MATLAB; modularity; concern; cross-cutting concern; token; block; aspect-oriented programming; aspect mining; aspect; refactoring; SQL; LARA.

# Resumo

Sabe-se que o apoio fornecido pelo MATLAB para a decomposição modular é limitado. Tais limitações dão origem a sintomas no código, que podem ser explorados por técnicas para a detecção de concerns, nomeadamente, concerns não modularizados. Trabalhos recentes na área de detecção de concerns em sistemas MATLAB identificaram vários padrões de código recorrentes que podem ser associados à presença de concerns específicos. Alguns dos concerns detectados mostraram-se não modularizados, isto é, atravessam a decomposição modular do sistema MATLAB.

As técnicas já desenvolvidas para a detecção de concerns não modulares em sistemas MATLAB ainda carecem de precisão e exatidão. Conforme proposto em trabalhos anteriores, as técnicas e ferramentas para identificar e representar padrões para a detecção de concerns precisam de maturação.

Esta tese contribui com uma estrutura mais precisa para representar bases de código MATLAB num repositório inteligente para código MATLAB desenvolvido antecedentemente. Aperfeiçoa a estrutura que representa o código MATLAB no qual o repositório se baseia, refinando a noção de bloco de código, e coleta padrões de código identificados em publicações anteriores, agregando-os num catálogo. Posteriormente, é feito um estudo preliminar sobre a aplicação dos blocos de código na detecção de concerns, validando padrões e concerns previamente estudados e avaliando a existência de novos.

**Palavras-chave:** MATLAB; modularidade; concern; concern não modular; símbolo; bloco; programação orientada a aspectos; mineração de aspetos; aspecto; refabricação; SQL; LARA.

# Contents

# List of Figures

# List of Tables

# Listings

# Glossary

| | |
|---|---|
| aspect | A modular representation of a cross-cutting concern. |
| concern | A program functionality or set of coherent operations. |
| cross-cutting concern | A concern that cuts across the modular structure of a program. |
| module | Set of routines that make up unique and independent operations. |
| token | A lexical element extracted from code. |
| weaver | Takes instructions specified by aspects and generates the final implementation code. |
| word-token | A string lexical element extracted from code. |

# Acronyms

| | |
|---|---|
| AOP | Aspect-Oriented Programming. |
| AST | Abstract Syntax Tree. |
| CCC | Cross-Cutting Concern. |
| DFT | Discrete Fourier Transform. |
| DSL | Domain-Specific Language. |
| ER | Entity Relation. |
| IR | Intermediate Representation. |
| OOP | Object-Oriented Programming. |
| PDG | Program Dependence Graph. |
| PMI | Pointwise Mutual Information. |
| SOM | Self-Organizing Map. |
| UbiSOM | Ubiquitous Self-Organizing Map. |

# Chapter 1

# Introduction

*This chapter focuses on introducing this thesis. It starts with a summary of the topic assessed in this thesis and the premises that motivated this study. Following, a brief description of the approach used and its objectives. It concludes with an overview of the remaining structure of the document.*

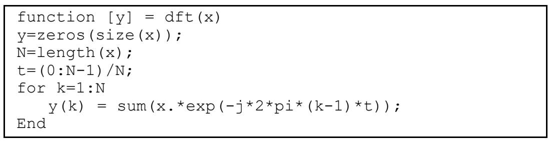
## 1.1 Problem and Motivation

Analysing and breaking a complex problem or system into parts that are easier to understand, program, and maintain helps to attain a clear *separation of concerns* at the source code level. A *concern* is a concept or cohesive set of functionalities that, along with more concerns, define the overall behaviour of a system. *Modularity* is the degree to which a software system can be divided in modules. Ideally, a system should enclose each concern in its own module, however programming languages come with some level of modularity limitations [35, 41].

Bruntink [5] states that *no matter how well a software system is decomposed into modular units, some concern will always cross cut its modular decomposition*, in other words, co-occurrence of concerns within the same module will continue to exist in software systems, restricting the extent to which one can benefit from the advantages of modularization. Concerns that cut across a modular decomposition of a software system, or unmodularized concerns, are named *cross-cutting concerns* (CCCs). These types of concerns show symptoms of being tangled with other concerns in the same module or scattering across multiple modules. Concern detection techniques explore source code for the identification of these code symptoms, consequently evaluating the presence of unmodularized concerns [16, 31].

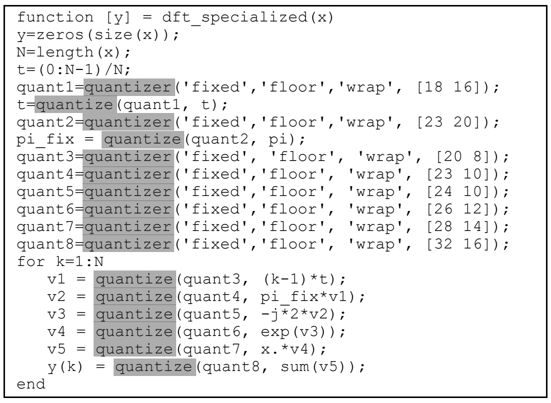
MATrixLABoratory (MATLAB) is a procedural programming language designed with the intention of reflecting the mathematical language and enable fast system development. Until recently, MATLAB was overlooked by cross-cutting concern detection and modularization studies. Despite the growing attention, current concern detection approaches still lack much needed efficiency, keeping the door open for further research [31].

Examples of polluted code and bad coding practices as the one presented in figures 1.1a and 1.1b have guided and motivated previous studies on concern detection in MATLAB. This example shows that MATLAB users tend to develop different versions of the same functions with little differences between them, mostly to adapt the execution of those functions improving the cost and efficiency of the hardware used to run them, which can be an indication of bad coding practices [13].



```
function [y] = dft(x)
y=zeros(size(x));
N=length(x);
t=(0:N-1)/N;
for k=1:N
   y(k) = sum(x.*exp(-j*2*pi*(k-1)*t));
End
```

(a) Code example needed to model a simple DFT



```
function [y] = dft_specialized(x)
y=zeros(size(x));
N=length(x);
t=(0:N-1)/N;
quant1=quantizer('fixed','floor','wrap', [18 16]);
t=quantize(quant1, t);
quant2=quantizer('fixed','floor','wrap', [23 20]);
pi_fix = quantize(quant2, pi);
quant3=quantizer('fixed', 'floor', 'wrap', [20 8]);
quant4=quantizer('fixed','floor', 'wrap', [23 10]);
quant5=quantizer('fixed','floor', 'wrap', [24 10]);
quant6=quantizer('fixed','floor', 'wrap', [26 12]);
quant7=quantizer('fixed','floor', 'wrap', [28 14]);
quant8=quantizer('fixed','floor', 'wrap', [32 16]);
for k=1:N
   v1 = quantize(quant3, (k-1)*t);
   v2 = quantize(quant4, pi_fix*v1);
   v3 = quantize(quant5, -j*2*v2);
   v4 = quantize(quant6, exp(v3));
   v5 = quantize(quant7, x.*v4);
   y(k) = quantize(quant8, sum(v5));
end
```

(b) Code example needed to model specialized fixed-point bit-widths to parcels of the DFT

Figure 1.1: MATLAB code examples for implementing two versions of a Discrete Fourier Transform (DFT) function [6]

Source code can be studied and manipulated in many ways and enable extraction of diverse amounts of information that can be used for detecting those little differences that add extra functionalities (or concerns) to the code (Chapter 3). Figure 1.1b presents an implementation of the DFT function with extra code to model specialized fixed-point bit-widths to parcels of the DFT, which can be identified by the *tokens* (i.e. lexical elements extracted from the source code) ’quantizer’ and ’quantize’, an example showing an additional unmodularized concern entangled with the main concern, the DFT. Concern detection techniques rely on understanding the programming language’s syntax and modular structures, and plenty of room for research and investigation is still available for techniques focused on MATLAB systems [6, 31].

## 1.2 Approach

This thesis proposes to follow up on a token-based approach for concern detection in MATLAB sources proposed by Monteiro et al. [31] (Section 4.2). The approach relies on the extraction of tokens with help of a tool, CCCExlorer. This tool developed for the approach had two main objectives: tokenize MATLAB source code repositories; extract token metrics from those repositories. The approach also associates specific groups of tokens to specific concerns, in which case patterns of occurrence of such tokens could be used to detect the presence of corresponding concerns or CCCs.

CCCExplorer is a metrics-extraction tool for MATLAB mainly based on code tokenization capabilities used by Relvas et al. for the development of *an intelligent repository for MATLAB code* [34]. The intelligent repository is an SQL database that accommodates code and concern data in one place, enabling the extraction and study of more complex information from the code repository [8].

In this thesis, MATLAB is studied at a source code level and the token-based tools are studied and revised. Sets of files or lines of code have been used as contexts from which metrics were extracted for consequent evaluation of the presence of concerns. This thesis focuses on how MATLAB systems can be organized in sets of *code blocks*, how metrics can be extracted from those code block structures and how those metrics can be indicators of the presence of concerns.

## 1.3 Objectives

This thesis aims to evaluate if MATLAB code blocks can be indicators of the presence of CCCs and also how they indicate it based on token metrics extracted from within each type of block. Consequently, it is expected to validate previous conclusions and results related to the token-based approach, with an emphasis on *schizophrenic functions* (Subsection 4.2.5). Also, it contributes with a code block context that is expected to improve the representation of MATLAB code repository in the database, in view of facilitating future studies following the present work. Ideally, it will develop further and more complex patterns beyond tokens and their relation to concerns. The objectives for this thesis are as follows:

- Study the MATLAB language from a structural, token-based perspective, evaluating how the language is structured in blocks and hierarchies of nested blocks, as the role of each type of block in MATLAB systems;
- Study the token-based approach and the tools (CCCExplorer and Intelligent Repository) developed in previous studies and adapt them to the problem of concern detection using MATLAB code blocks;
- Refine the concept of MATLAB code block, conceptualize the structures needed to handle MATLAB code blocks and implement them in the token-based tools;

- Tokenize, extract metrics, evaluate results and study the presence of unmodularized concerns in a MATLAB open source code repository using the revised tools;

## 1.4 Document Structure

This document addresses the MATLAB language and the detection of concerns as two complementary subjects, firstly analysing the language for, in the second, studying the detection of cross-cutting concerns. Then follows with a study focused on modularity issues extant in MATLAB. Subsequent to the current chapter, the structure of the document is as it follows:

- **Chapter** 2 is an overview of the MATLAB language, starting with a short description of its uses and history. It then follows with an introduction to the programming language's syntax and components it provides to represent and support modules, namely how the code can be structured from a perspective of blocks and hierarchies of nested blocks.

- **Chapter** 3 overviews concepts related software modularity and concern modularization. It starts by introducing some of the issues that motivated past studies of software modularity, followed by descriptions and examples of some of the concepts accessed in this thesis.

- **Chapter** 4 focuses on the software modularity problem within the MATLAB language, describing the state of the art on MATLAB aspect mining and AOP. It presents a few CCC examples that motivated previous studies on aspect mining and refactoring of MATLAB systems, followed by tools and methods developed in a token-based approach.

- **Chapter** 5 describes the motives that guided the extension and implementation of blocks of code into the previously mentioned tools. Then it discusses the system (CCCExplorer) refactoring decisions to enable the implementation of the extension (code blocks) into the system, providing technical information about the system, design decisions and structure.

- **Chapter** 6 analyses the MATLAB repository from the perspective of blocks of code with the help of the developed extension to the intelligent repository. It presents a brief overview of the repository and, based on the tailored notion of CCC in MATLAB, it follows with an analysis on each type of block and the particular case of schizophrenic functions.

- **Chapter** 7 presents a summary of the work conducted during this thesis, including results obtained and opportunities for future work.

# Chapter 2

# MATLAB

*The following chapter is an overview of the MATLAB language, starting with a short description of its uses and history. It then follows with an introduction to the programming language's syntax and components it provides to represent and support modules, namely how the code can be structured from a perspective of blocks and hierarchies of nested blocks.*

## 2.1 About MATLAB

The MATLAB programming language was originally designed as an interactive interface to numerical libraries and had only one data type, matrices. Over the years the language has been extended and used for substantial programming projects, becoming on of the most popular dynamic programming languages [13].

MATLAB has a very large and increasing user base world-wide, comprising scientists, engineers and students. In 2018 it was estimated that its user base counted on aprox. 3 million users. Also, it has the support of a large and increasing community, currently with 365.000 contributors and a daily activity of 120 answers, 25.000 downloads and 730 solvers. It is one of the key languages used in education, research and development for scientific and engineering applications. There are currently over 2000 books based on MATLAB and its companion software, Simulink, for teachers, students, and professionals. The large and increasing collection of books reflects the growing use of these tools for research and development within many scientific areas [9, 26].

MATLAB is a numerical computing environment, designed for fast prototyping and quick application development. However, this makes it unpractical to write in-depth code and has a negative impact on developing reliable and reusable programs, and negative a impact on performance [13].

## 2.2 Syntax

### 2.2.1 Variables

MATLAB variables do not have type declaration and they have only one data type, which are matrices. For example, a string in Java is the equivalent of a character array in MATLAB. Variables can either be global or local. They are usually local variables, which means that they can only be accessed in the function where they are defined. In the case of global variables, those variables are declared once and all the functions that call it, share a single copy of the variable. There are some example variable value assignments displayed in Listing 2.1 [23].

Listing 2.1: Value assignment examples

```matlab
% Numeric value
x = 1;

% String value
myText = 'Hello World';

% Array with four elements
a = [1 2 3 4];

% Matrix 3-by-3
m = [1 2 3; 4 5 6; 7 8 10];

% Function value assignment to f
f = myfun();
```

### 2.2.2 Matrices and Arrays

Arrays are matrices that only have one column and each cell of the array is indexed and can store any type of data. One can also declare arrays in different ways as presented in Listing 2.2 [23].

Listing 2.2: Array declaration examples

```matlab
% Array with four elements
a = [1 2 3 4]

% Matrix 3-by-3
a = [1 2 3; 4 5 6; 7 8 10]

% Matrix 5-by-1 of zeros
z = zeros(5,1)

% Matrix 2-by-2 of complex numbers
c = [3+4i, 4+3j; -i, 10j];
```

### 2.2.3 Operators

Operators are symbols that tell the compiler to perform a certain mathematical or logical operation. Each operator symbol has its correspondent function that acts as if using the operator. These operators and elementary operations can be divided into five different groups [24]:

- **Arithmetic Operations** are array type operations that perform element-by-element operation, or matrix type operations, following the rules of linear algebra (Table 2.1);

Table 2.1: Arithmetic Operations

| | Opt. Array | Opt. Matrix | Func. Array | Func. Matrix |
|---|---|---|---|---|
| Addition | A+B | A+B | plus(A,B) | plus(A,B) |
| Subtraction | A-B | A-B | minus(A,B) | minus(A,B) |
| Multiplication | A.*B | A*B | times(A,B) | mtimes(A,B) |
| Right Division | B./A | B/A | rdivide(A,B) | mrdivide(B,A) |
| Left Division | A. | A | ldivide(B,A) | mldivide(A,B) |
| Exponentiation | A.^B | A^B | power(A,B) | mpower(A,B) |
| Transpose | A.’ | A’ | transpose(A) | ctranspose(A) |

- **Relational Operations** perform element-by-element comparisons between arrays with the same size or an array with a scalar (Table 2.2);

Table 2.2: Relational Operations

| | Operator | Function |
|---|---|---|
| Less than | A<B | lt(A,B) |
| Greater than | A>B | gt(A,B) |
| Less than or equal to | A<=B | le(A,B) |
| Greater than or equal to | A>=B | ge(A,B) |
| Equal to | A==B | eq(A,B) |
| Not equal to | A~=B | ne(A,B) |
| Array equality | - | isequal(A,B, ...) |
| Array equality (treating NaN values) | - | isequaln(A,B, ...) |

- **Logical Operations** check if a condition was fulfilled, returning 0 (false) or 1 (true) (Table 2.3);
- **Set Operations** are used to perform joins, unions and intersections between two arrays;
- **Bit-Wise Operations** are used to set, shift or compare a specific bit/value in one array.

Table 2.3: Logical Operations

| | Operator | Function |
|---|---|---|
| Logical AND | A&B | and(A,B) |
| Logical OR | A\|B | or(A,B) |
| Logical AND (with short-circuiting) | A && B | Logical Operators: |
| Logical OR (with short-circuiting) | A\|\|B | Short-Circuit && \|\| |
| Logical NOT | ~A | not(A) |

### 2.2.4 Statements and Statement blocks

Statements require that the programmer use one (or more) condition(s) to evaluate the code. Each statement block is identified by its corresponding keyword and all require the 'end' keyword for delimiting the block of code affected by the statement. One line of code can include several statement blocks, as long as they remain correctly delimited [25].

Conditional statements (Listings 2.3, 2.4 and 2.5) evaluate the condition and control the flow of the program. For both 'if' and 'switch', MATLAB executes the code corresponding to the first true condition, and then exits the code block [25].

Listing 2.3: If statement example

```matlab
% Generate a random number
a = randi(100, 1);

% If it is even, divide by 2
if rem(a, 2) == 0
    disp('a is even')
    b = a/2;
end
```

Listing 2.4: Elseif statement example

```matlab
yourNumber = input('Enter a number: ');
if yourNumber < 0
    disp('Negative')
elseif yourNumber > 0
    disp('Positive')
else
    disp('Zero')
end
```

Loop control statements (Listings 2.6 and 2.7) guarantee that the program executes the same block of code while the condition is evaluated as true. These statements allow the repetitive execution of a block of code, to create more complex algorithms in MATLAB [25, 28].

Listing 2.5: Switch statement example

```
[dayNum, dayString] = weekday(date, 'long', 'en_US');
switch dayString
   case 'Monday'
      disp('Start of the work week')
   case 'Tuesday'
      disp('Day 2')
   case 'Wednesday'
      disp('Day 3')
   case 'Thursday'
      disp('Day 4')
   case 'Friday'
      disp('Last day of the work week')
   otherwise
      disp('Weekend!')
end
```

Listing 2.6: For statement example

```
% conventional for-loop
x = ones(1,10);
for n = 2:6
    x(n) = 2 * x(n - 1);
end

% for-loop executed in parallel on M workers (threads)
M = 1;
y = ones(1,100);
parfor (i = 1:100,M)
     y(i) = i;
end
```

Listing 2.7: While statement example

```
n = 1;
nFactorial = 1;
while nFactorial < 1e100
    n = n + 1;
    nFactorial = nFactorial * n;
end
```

The 'try-catch' statement (Listing 2.8) is used to catch errors occurring after the 'try' keyword and before the 'catch' keyword and specify the alternative behaviour after the 'catch' keyword, overriding the default error behavior for a set of program statements. If any statement in a 'try' block section generates an error, program control goes to the 'catch' block section, which usually contains error handling statements [22].

Listing 2.8: Try-catch example

```
% Catch any exception generated by calling the nonexistent function,
    notaFunction.
% If there is an exception, issue a warning and assign the output a value
    of 0.
try
    a = notaFunction(5,6);
catch
    warning('Problem using function.  Assigning a value of 0.');
    a = 0;
end
```

### 2.2.5 Functions and Function blocks

There are several types of functions available with MATLAB, including local functions, nested functions, private functions, and anonymous functions. For readability, the 'function' and 'end' keyword are used to delimit the code of each function in a file. The 'end' keyword is required when: any function in the file contains a nested function; the function is a local function within a script file (Subsection 2.4.1); the function is a local function within a function file, and any local function in the file uses the 'end' keyword [29].

- **Anonymous functions** are functions that are not stored in a program file, but are associated with a variable whose data type is of function_handle. Anonymous functions can accept inputs and return outputs, just as standard functions do. However, they can contain only a single executable statement.

Listing 2.9: Anonymous functions example

```
% Creating a handle to an anonymous function that
% finds the square of a number.
sqr = @(x) x.^2;

% Integral of the sqr function from 0 to 1, passing
% the function handle to the integral function.
q = integral(sqr,0,1);
```

- In a function file, the first function in the file is called the main function. This function is visible to functions in other files, or you can call it from the command line. Additional functions within the file are called **local functions**, and they can occur in any order after the main function.

Listing 2.10: Local functions example

```
% Main function
function [avg, med] = mystats(x)
    n = length(x);
    avg = mymean(x,n);
    med = mymedian(x,n);
end

% Local function mymean
function a = mymean(v,n)
    a = sum(v)/n;
end

% Local function mymedian
function m = mymedian(v,n)
    w = sort(v);
    if rem(n,2) == 1
        m = w((n + 1)/2);
    else
        m = (w(n/2) + w(n/2 + 1))/2;
    end
end
```

- **Nested functions** are functions that are syntactically enclosed within a parent function. Any function in a program file can include a nested function.

Listing 2.11: Nested function example

```
function parent
disp('This is the parent function')
nestedfx

   function nestedfx
      disp('This is the nested function')
   end
end
```

- **Private functions** are useful for limiting the scope of a function. A function is private when stored in a sub-folder with the name 'private'. This makes the functions visibility limited to functions or scripts in the folder immediately above the private sub-folder.

## 2.3 Object-Oriented MATLAB

Object-oriented programming promises to increase MATLAB's modularity, allowing for code maintenance reduction and an improvement to code reusability, scalability, reliability and flexibility. Major enhancements have been made to MATLAB since 2008 regarding object-oriented programming to deal with its modularity issues [27].

Creating classes can simplify programming tasks that involve specialized data structures or large numbers of functions that interact with particular kinds of data. MATLAB classes support function and operator overloading (i.e. different functions/operators have different implementations depending on their arguments), controlled access to properties and methods, reference and value semantics, and events and listeners [9].

Object-oriented programming in MATLAB involves using: class definition files, enabling definition of properties, methods, and events; classes with reference behavior, aiding the creation of data structures such as linked lists; events and listeners, allowing the monitoring of object property changes and actions. For this purpose, MATLAB organizes class definition in five modular blocks, each identified and delimited by its own keyword and the termination with the 'end' keyword (see detailed example of a class definition in Listing I.1) [27].

### 2.3.1 Class definition blocks

Class definition blocks are identified with the 'classdef' keyword. A 'classdef' block contains the class definition and the specification of the class's attributes and superclasses. The 'classdef' block also contains the 'properties', 'methods', 'events' and 'enumeration' subblocks (Listing 2.12) [27].

Listing 2.12: Class definition syntax - Properties methods and events

```
classdef (Attributes) ClassName < SuperclassName
   properties (Attributes)
      PropertyName
   end
   methods (Attributes)
      function obj = methodName(obj,arg2,...)
         ...
      end
   end
   events (Attributes)
      EventName
   end
end
```

### 2.3.2 Properties blocks

A 'properties' block defines properties having the same attribute settings. A class block may have multiple 'properties' blocks with different attribute settings. The 'properties' block can specify a default value for each property individually, assign property values in a class constructor, define properties with constant values, assign property attribute values on a per block basis, define methods that execute when the property is set, define the class and size of property values and define properties that do not store values, but whose values depend on other properties (Listing 2.12) [27].

### 2.3.3 Methods blocks

Class definitions can contain multiple 'methods' blocks, each specifying different attribute settings that apply to the methods in that particular block. Methods are functions that implement the operations performed on objects of a class and are specified in the 'methods block'. Methods, along with other class members, support the concept of encapsulation, i.e. class instances contain data in properties and class methods operate on that data (Listing 2.12) [27].

### 2.3.4 Events blocks

Events are notifications that objects transmit as a response to something that happens, such as changing a value of a property or a user interaction with an application. Listeners execute functions when notified that the event of interest occurs. Class definitions can contain more than one 'events' block each with a different attribute settings specification (Listing 2.12) [27].

### 2.3.5 Enumeration blocks

Enumerations are used to represent fixed sets of named values, where all the values are of the same type. MATLAB recommends that enumerations should be put in separate classes, where a 'classdef' block would contain a single 'enumeration' block. These are called enumeration classes, which can be derived from other classes, inheriting arithmetic and ordering operations of the superclass (Listing 2.13) [27].

Listing 2.13: Class definition syntax - enumeration

```
classdef (Attributes) ClassName < SuperclassName
   enumeration
      EnumName
   end
end
```

## 2.4 M-Files

M-files are the MATLAB program files, identified with the .m extension, and they can be scripts, functions or classes. All types of m-files allow the reusing of sequences of commands by storing them in program files. Scripts are the simplest type of program, since they store commands exactly as you would type them at the command line. Functions are more flexible and more easily extensible and can be called in other m-files, if in accordance with MATLAB's file structuring rules. Classes define a set of objects with common traits and behaviors. A subclass defines both a subset of objects that exhibit the traits and behaviors defined by the superclass and additional traits and behaviors not exhibited by all instances of the superclass [27, 29].

### 2.4.1 Script files

Scripts are the simplest kind of program file because they have no input or output arguments. They are useful for automating series of MATLAB commands, such as computations that have to be performed repeatedly from the command line or series of commands that need referencing. Sometimes they adopt the behaviour similar to a 'main' function in Java or C, becoming the entry point to the programs execution [29].

A script can include valid MATLAB expressions, control flow statements, comments, blank lines and function calls. When execution of the script completes, the variables remain in the MATLAB workspace [29].

### 2.4.2 Function files

Function files allow for a practical structuring of a MATLAB program. Instead of having all operations in a single file there is the possibility of separating each functionality in separate function files. This facilitates the interpretation and editing of the code, as well as the programs maintainability, which are similar symptoms to the use of objects in object-oriented programming (OOP) [29].

The body of a function can include valid MATLAB expressions, control flow statements, comments, blank lines and nested functions. Any variables that you create within a function are stored within a workspace specific to that function, which is separate from the base workspace, opposite to scripts [29].

Program files can contain multiple functions. If the file contains only function definitions, the first function is the main function, and is the function that MATLAB associates with the file name. Functions that follow the main function or script code are called local functions. Local functions are only available within the file [29].

### 2.4.3 Class files

Class files describe the characteristics shared by a set of objects. The values contained in an object's properties are what make an object different from other objects of the same class. The functions defined by the class, or methods, are what implement object behaviors that are common to all objects of a class [27].

There are two ways to make class files detectable by the workspace environment: a folder that is on the MATLAB path; a folder inside a path folder named with the '@' character and the class name (Figure 2.1) [27].

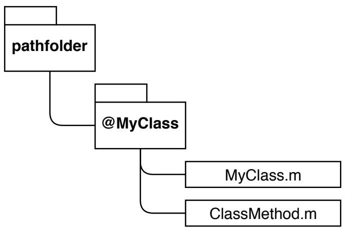


Figure 2.1: Folder organization example

An object is an instance of a class. When a program executes, the object is created based on its class and behaves in the way defined by the class. The values stored in MATLAB variables all belong to a class. These values include not only what you might normally consider objects, such as a time series or state space object, but also simple doubles [27].

Mathworks claims that building MATLAB applications using OO techniques leads to robust and maintainable applications for others to use and integrate with other related applications throughout an organization [27].

## 2.5 Toolboxes

Toolboxes are a set of functions designed for a related purpose provided as packages. A toolbox is composed of several function files merged into a single file represented by the file-name extension .mltbx [20].

For example, Parallel Computing Toolbox helps solving computationally and data-intensive problems using multi-core processors. It adds more functions and statement blocks for the purpose, such as 'parfor', which behaves like a 'for' statement but for multi-threaded processing, and 'spmd', a statement block that delimits the code to be processed in parallel. Toolboxes can be purchased as enhancements for the MATLAB programming environment. However, in the context of this thesis, they will be referred as a set of m-files, with the .m file extension, associated to the same project folder.

## 2.6 Summary

MATLAB is a fast prototyping language that is intuitive and easier to learn thanks to its similarity to mathematical logic. Like any other programming language, MATLAB system developers rely on the languages components in order to create mathematical tools, functionalities and features. This study focuses on MATLAB source code as a set of blocks and hierarchies of nested blocks, showing MATLAB from a perspective of structures of blocks. Statement blocks allow the development of more complex algorithms and can be found on any type of MATLAB source files. Blocks related to classes or functions provide better modular structures to build MATLAB systems.

Chapter 3

# Concerns and Software Modularity

*This chapter overviews concepts related software modularity and concern modularization. It starts by introducing some of the issues that motivated past studies of software modularity, followed by descriptions and examples of some of the concepts accessed in this thesis.*

## 3.1 What is Modularity?

In software engineering, decomposing a large software system into smaller parts is an essential way of facilitating the management and evolution of complex software systems. Separating system functionalities in modules facilitates independent simultaneous work on the same system, team specialization, localized change, systematic testing and quality assurance, and work planning [40].

Modularity is the degree to which a system's components may be separated and recombined. The concept of modularity is mainly used to reduce complexity by dividing a system into varying degrees of interdependence and independence and hiding the complexity of each part behind an abstraction and interface [3].

Tarr et. al. point out that we can decompose a software system in small modular components, however, some functionality will always cut across the module decomposition. In other words, some functionality will persist in not being captured cleanly within a single module and, consequently, its code will be scattered throughout other modules, which is a clear indicator of limitations in the system's modular decomposition. In an ideal scenario, a single concern is what is intended to be captured in a single module [39].

## 3.2 Concerns

A *concern* is defined as any abstraction, concept or cohesive set of functionalities that is ideally enclosed in its own module, for the sake of comprehensibility, ease of maintenance and evolution. It can be a feature, functionality, property, requirement or, for example, as in OOP, an object. Essentially, a program consists of multiple concerns that define its overall behaviour [31, 32, 41].

The Java code example in Listing 3.1 depicts the main concern of class Point, related to the variables, functions and/or lines of code that deal with the values of x and y. These variables represent the coordinates of an object of class Point in the 2-dimensional space. It is also represented another concern that addresses, as the variable name suggests, the display of an object of class Point in the 2-dimensional space, which is highlighted in grey.

Listing 3.1: Point class [31]

```java
public class Point implements FigureElement {
    private int x, y;
    private Display display;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public Point(int x, int y, Display display) {
        this(x, y);
        setDisplay(display);
    }
    public void setX(int x) {
        this.x = x;
        display.update(this);
    }
    public void setY(int y) {
        this.y = y;
        display.update(this);
    }
    public void setDisplay(Display display) {
        this.display = display;
    }
    public void moveBy(int dx, int dy) {
        x += dx; y += dy;
        display.update(this);
    }
}
```

These concerns are easily identified, however, they coexist in the same modular structure that is the class Point. Ideally, both concerns should be decomposed in their respective modules, being for example, separated in two classes: one deals with the coordinate values; the other is responsible for displaying the coordinates in a 2-dimensional space.

## 3.3 Cross-Cutting Concerns

*Cross-cutting concerns* (CCCs) are concerns that do not align with the primary decomposition and tend to cut across the decomposition units. Most CCCs cannot be cleanly decomposed from the rest of the system in both the design and implementation. This is due to the lack of modularization capabilities of programming languages regarding some concerns. The usual symptoms of the presence of a CCC in source code are code scattering and code tangling, which are explained further in section 3.4 [31, 43].

In the example code in Listing 3.1 is highlighted in grey the display concern, which is tangled with the main concern of class point. Knowing that the display concern presents an entanglement symptom, it is designated as a CCC.

## 3.4 Scattering and Tangling

*Tangling* is a phenomenon that is observed when other concerns are mixed in the code with the main concern, which makes it difficult to read and comprehend all the module's concerns. An extreme case of the presence of this symptom is presented in Figure 1.1b. *Scattering* refers to code regarding some concern that is fragmented throughout several source files, sometimes related to sections of code that are similar, placed in different locations but with little differences [31].

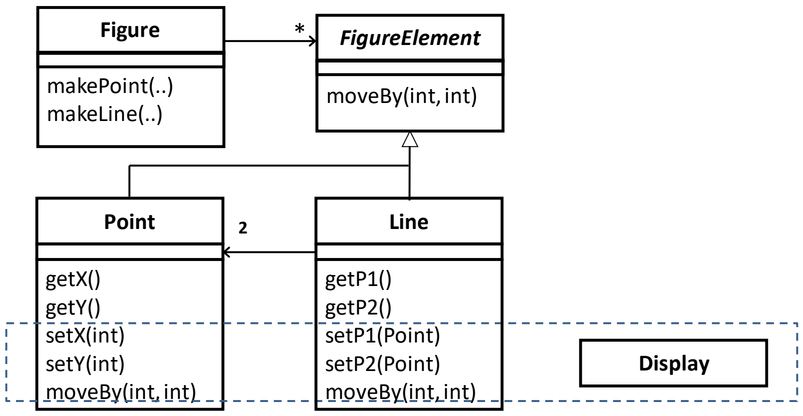


Figure 3.1: Figure elements class schema [31]

To better understand what scattering is, the example in Figure 3.1 is analyzed. Considering that the display concern would also be present in another class, for instance, a class 'line' which is also a figure element. Assuming a programmer would want to change the 'update' method - change the method call names from 'update' to 'render' and to include some extra information. These changes become a nightmare since the display concern is scattered across the system's code base. In more complex software systems there could be hundreds of places where this change would have to be made instead of just one.

## 3.5 Aspect Mining and Aspect-Oriented Programming

*Aspect mining* is a fundamental activity that software developers and maintainers perform and results in the identification and location of CCCs as latent aspects of the code. An *aspect* is a modular representation of a CCC that otherwise would not be modularized, depending on the system, mechanism or language available. Aspects tend to be properties that affect the performance or semantics of the components in systemic ways rather than units of the systems functional decomposition [16, 31].

The display concern from Listing 3.1 is an example of a CCC. Although aspect mining techniques are used to detect the display concern, it does not necessarily need to be modularized using aspects. Aspects are complementary composite modules that serve as extensions to the system's programming language, containing integral specifications that implement complementary functionalities of the system. The display concern implemented using aspects would be as simple as writing the specifications needed to insert the code regarding the concern in the correct lines, but it could also be implemented using the Java language's capabilities. Imagining that there is additional code in the example to specify unit types, changing how the value is stored for each coordinate, x and y, and the specification is different between each method. This code would add an additional concern regarding data type specifications, cutting across the module's main concern, therefore another CCC. Such a concern would probably need the help of aspects to be modularized [18, 31].

There are several aspect mining techniques that allow for the location and identification of CCCs 3.5.1. Typically, software developers and maintainers try to locate source code related to a concern using a variety of ad-hoc techniques such as scrolling through files, following call graph links, analyzing dynamic information or searching files using mechanisms that perform repetitive searching tasks[31, 36].

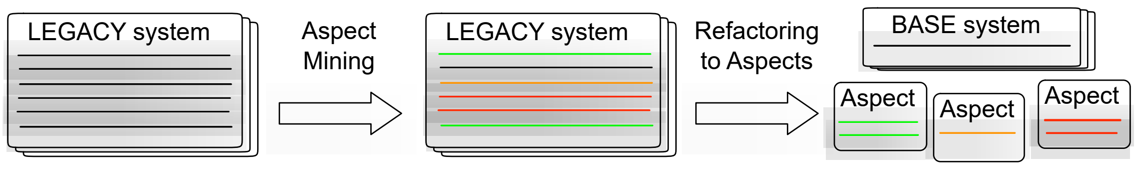


Figure 3.2: Migrating a legacy system to an aspect-oriented system [16]

Aspect-Oriented Programming (AOP) came up in the 90s as a paradigm aimed at making CCCs explicit using code generation techniques designed to take instructions specified by aspects, generate the final implementation code and including it in the application logic. This was proposed as an approach to modularize CCCs and thus eliminate the negative symptoms of scattering and tangling in systems [6, 13, 14, 16, 31, 40].

### 3.5.1 Cross-Cutting Concern Identification

Aspect mining techniques seem to have in common the search for symptoms of CCCs, using either techniques from data mining and data analysis, like formal concept analysis and cluster analysis, or more classic code analysis techniques, like program slicing, software metrics and heuristics, clone detection and pattern matching techniques, dynamic analysis, and so on. Bruntink et al. evaluated the suitability of clone detection techniques that deal with source code of the system, data that is acquired by executing or manipulating the code, for the identification of CCCs [5, 16]:

- Text based - perform little or no transformation to the 'raw' source code before attempting to detect identical or similar (sequences of) lines of code. Typically, white space and comments are ignored;
- Token based - apply a lexical analysis (tokenization) to the source code, and subsequently use the tokens as a basis for detection (technique application in previous work carried out by Monteiro et al. [32]);
- AST based - use parsers to first obtain a syntactical representation of the source code, typically an *abstract syntax tree* (AST). The detection algorithms then search for similar sub-trees in this AST;
- PDG based - go one step further in obtaining a source code representation of high abstraction. *Program dependence graphs* (PDG's) contain information of semantical nature, such as control and data flow of the program (technique application in previous work carried out by Shepherd et al. [36]);
- Metrics based - are related to hashing algorithms. For each fragment of a program the values of a number of metrics is calculated, which are subsequently used to find similar fragments (technique application in previous work carried out by Monteiro et al. [31]).

Data extracted from these techniques can also be used for developing more sophisticated aspect mining techniques, using machine learning algorithms, as already studied by Marques et al. [8]

### 3.5.2 Cross-Cutting Concern Modularization

As previously implied, conventional programming techniques lack modularization capabilities for some CCCs, which lead to the development of aspect-oriented tools to help with that problem, such as language extensions and aspect-oriented features [13, 17].

The most well known aspect-oriented languages are AspectJ [2], for Java, and AspectC++ [38], for C++. They serve as extensions of the programming language and provide support for modular implementation of a range of CCCs. MATLAB has also been a study subject to aspect-oriented extensions. Aslam et al. contributed with Aspect-Matlab [1], a novel aspect-oriented language that serves as an extension to MALTLAB. These three extensions are language dependent, meaning that aspects written with these extensions can only be applied to each ones respective language.

However, Cardoso et al. contributed with LARA [7], a Domain-Specific Language (DSL) for source-to-source transformations and analysis, inspired by AOP concepts. LARA explores the idea of the possibility of having a single AOP language capable of selecting points of interest and apply source code transformations agnostic to the target language. LARA aspects are applied using weavers, translating the abstract concerns described in LARA aspects to a concrete language [4].

When writing an aspect with these language extensions, the structure of a CCC is expected to be more clear and easy to understand. Aspects also yield a more modular structure, making it possible to develop implementations of crosscutting functionality that can be plugged and unplugged into the system [18].

## 3.6 Summary

No matter how well structured a software system is, some functionality, or concern, will persist on not being enclosed in its own module. This tenacity comes with many problems that motivate the development of techniques for solving those problems. Existing aspect mining and concern modularization techniques focused on the MATLAB language still have some limitations, making room for further researching and expansion of those techniques.

Chapter 4

# A Study on Modularity in MATLAB

*The following chapter focuses on the software modularity problem within the MATLAB language, describing the state of the art on MATLAB aspect mining and AOP. It presents a few CCC examples that motivated previous studies on aspect mining and refactoring of MATLAB systems, followed by tools and methods developed in a token-based approach.*

## 4.1 Cross-Cutting Concerns in MATLAB

Currently aspect mining is well advanced for systems that follow the OOP paradigm, but for procedural languages like MATLAB it is a different situation. Due to the specific characteristics and different typical uses of MATLAB, a need to rethink the notion of CCC emerged, which also required fresh approaches for their identification in MATLAB code bases. It should be noted that object-oriented MATLAB (Section 2.3) is a technology that still needs maturing [31].

Figure 1.1a presents the code needed for the implementation of the Discrete Fourier Transform (DFT) algorithm, widely used in signal processing systems. Figure 1.1b another implementation of the same DFT algorithm but with specialized fixed-point bit-widths to different parameters (or sets of parameters) of the algorithm. This is a common practice for MATLAB users since these specifications are usually needed to satisfy various requirements, namely low power dissipation, low energy consumption, better performance and fewer use of hardware resources, therefore, it is very common to hold different versions of the same algorithm [6].

As implied in sections 3.2, 3.3 and 3.4, a single module should hold a single concern, otherwise other concerns might cut across the module's main concern. These other concerns are CCCs, since are not enclosed within their own module. The example in figures 1.1a and 1.1b is an extreme case of code polluted with a secondary concern, the data type specification of each portion of the DFT function, and cramps the comprehension of the function itself. This concern can be identified by the functions 'quantizer' and 'quantize' that are used for specializing fixed-point bit-widths in MATLAB.

Figure 4.1 shows an example of a MATLAB function in two versions where the value returned for a given parameter value is the same in both cases. At the left a clean version is shown, whose code relates to its core concern exclusively, which is to compute the result of the exponential function applied to a specific parameter value using the first N terms of the power series expansion. At the right, a tangled version is shown also including a Visualization concern (see entry 7. in Table 4.1), which in this case means preparing a call to function 'plot', since 'plot' pops open a new window with the plotted data. The code pertaining to this secondary concern is highlighted in grey. The extra code is mostly about building the vector data required for building a two-dimensional representation of the function within a given range, which will feed the 'plot' function at the end [31].

```
function z = expo(x,n)
  y = 1;
  for i = 1:n
    y = y + x^i/factorial(i);
  end
  z = y;
```

```
function z = expo(x,n,p)
  P(1) = 1;  Y(1) = 1;
  for i = 1:n
    P(i+1) = P(i)+1;
    Y(i+1) = Y(i) + x^i/factorial(i);
  end
  z = Y(n+1);
  if (p) plot(P,Y) end
```

Figure 4.1: Illustrative example of a CCC in MATLAB [31]

The tangled version defines an additional (and of optional use) parameter to support a choice between creating and not creating the plot representation. The sole motivation for this parameter, like the distinct constructors in the Java example from Listing 3.1, is support to one of the primary advantages of modularity: (un)pluggability of the functionality concerned [31].

## 4.2 A Token-based Aspect Mining Approach

Monteiro et al. describe an exploratory, use of the token-based approach to concern detection for MATLAB systems [32]. They propose a notion of CCC tailored for the specific characteristics of MATLAB code bases. An analysis of data obtained from a tool, CCCExplorer, using the approach in a 35 thousand MATLAB m-file repository indicated that the approach was valid for detecting several kinds of CCCs in MATLAB systems.

### 4.2.1 CCCExlorer Tool

CCCExplorer is a tool written in Java and it was developed and introduced by Monteiro et al. [31] as a concern detection, token-based aspect mining tool in MATLAB systems. Initially, it parsed and decomposed MATLAB source files into sequences of tokens, i.e., lexical elements (words, punctuation and grammar rules of the language) extracted from the code, and allowed the computation of metrics based on those tokens.

The need for maturing the concern detection technique raised and it has been reflected in the evolution of CCCExplorer. Now, it can generate a series of SQL insert commands to create an SQL database that reflects the m-file repository, settling it as *An intelligent repository for MATLAB code* developed by Relvas et al. [34]. This intelligent repository opens the door to studies on more complex aspect mining techniques.

### 4.2.2 M-file Repository

The m-file repository is composed of publicly available MATLAB files extracted from open source platforms like Sourceforge [30] and GitHub [12]. For the study conducted during this thesis, the need to acquire more recent m-files that could also contain OO MATLAB code examples emerged, increasing the previous 35 thousand m-file repository to approximately 65 thousand m-files.

This was achieved with an open source script, GitHub project fetcher, adapted to download projects that contain any m-files and delete any other files and empty folders (Scripts III.1, III.2 and III.3). This script may also be used to continuously enrich the intelligent code repository and take advantage from UbiSOM's data streaming capabilities.

### 4.2.3 Concern-token mapping

The extraction of lexical elements allowed a study where they were able to associate specific groups of tokens to specific concerns, in which case patterns of occurrence of such tokens could be used to detect the presence of the corresponding concerns [31, 32].

The tokens considered are mostly function names from MATLAB libraries and toolboxes, since they provide stronger guarantees that the meaning assigned to each token is consistent across the repository analysed, thus providing coherent associations to specific concerns. The analysis resulted in a first version of the concern-token association presented in Table III.1. During this thesis, the tokens included in this table are designated as *concern tokens* [32].

Table 4.1: Mapping between Concerns and corresponding Tokens revised [15]

| Concern | Sub-concern | | Tokens |
|---|---|---|---|
| 1. Verification of function arguments and return values | | | nargchk, nargin, nargout, nargoutchk, varargin, varargout |
| Data type handling | 2. Specialization | | double, fi, int8, int16, int32, int64, quantize, quantizer, single, uint8, uint16, uint32, uint64 |
| | 3. Verification | Identification and numeric types | cast, class, intmax, intmin, isa, isboolean, iscell, ischar, isfloat, isinf, isinteger, islogical, isnan, isnumeric, isobject, isreal, isstr, isstruct, realmax, realmin, typecast |
| | | Matrices and arrays | isempty, isfield, isrow, isscalar, isvector, length, ndims, numel, range, size |
| 4. Dynamic properties | | | eval, evalc, evalin, feval, inline |
| 5. Console messages | | | annotation, assert, disp, display, error, last, lastwarn |
| 6. Printing | | | orient, print, printdlg |
| 7. Visualization | 2D and 3D plots | Animation | getframe, movie |
| | | Contour plots | clabel |
| | | Data distribution plots | hist, histogram, scatter |
| | | Line plots | errorbar, fplot, gplot, loglog, plot, plot3, semilogx, semilogy |
| | | Polar plots | polar |
| | | Surfaces, volumes and polygons | mesh, meshgrid, surf |
| | Formatting and Annotation | Axes appearance | axis, box, datetick, figure, grid, plotyy, subplot |
| | | Titles and labels | legend, line, rectangle, text, title, xlabel, ylabel, zlabel |
| | Graphic objects | Identification | gca, gcbf, gcbo, gco |
| | | Properties | reset, set |
| | | Programming / output | cla, clf, close, hold, ishold, newplot |
| | Images | | frame2im, image, iminfo, imread, imwrite |
| | Visual exploration | | datacursormode, pan, rotate, rotate3d, zoom |
| 8. File I/O | | | diary, fgetl, fgets, fopen, fprintf, fread, fscanf, fwrite, hgload, hgsave, load, save, saveas, uisave |
| 9. System | Code analysis | | run, timerfind |
| | Control flow | | break, next, pause, wait, xbreak |
| | Data import and export | | clear, who, whos |
| | Dates and time | | addtodate, clock, date, etime, now, step |
| | Debugging | | dbstop, echo |
| | Entering commands | | ans, slist, stop |
| | Files and folders | | exist, rehash |
| | Functions | | inputname, isvarname, mfilename, mislocked, mlock, symvar |
| | Performance and memory | | cputime, memory, pack, profile, tic, toc |
| | Scripts | | batch, input |
| | Startup | | start |
| | Using external libraries | | calllib, libisloaded, loadlibrary, mex, mexext, unloadlibrary |
| 10. Memory allocation / deallocation | | | delete, global, ones, persistent, zeros |
| 11. Parallelisation | | | cancel, codistributed, codistributor, gather, labindex, labProbe, matlabpool, numlabs, parfor, pload, pmode, promote, resume, sparse, spmd, subsasgn, subsref |

For every file in the repository, a number of metrics was computed, such as the number of times a given token appears, the number of different tokens in each m-file, etc. These metrics allowed a top-level observation of the symptoms (scattering and tangling) within the MATLAB code base with insigth on Table III.1. It indicated a significant proportion of MATLAB code that comprised promising candidates for extraction to future aspectual extensions of MATLAB [31].

Jota [15] improved the table of concerns and concern tokens (Table III.1). Table validation was performed using pointwise mutual information metric (PMI) [44] as a linguistic computation method, excluding tokens that presented few occurrences, thus focusing the aspect mining technique in more relevant points of interest. Furthermore, it clarified and improved the token-concern relationships, resulting in Table 4.1, more focused on the m-file repository.

### 4.2.4 Token patterns

Concurrently with the validation of the token-concern relation table, Jota [15] also studied *token patterns* in m-files. The study was focused on calculating the PMI from the co-occurrence of pairs of tokens belonging to different concerns, also deeming more interesting candidates for detecting CCCs. Co-occurrence of concern tokens belonging to the same concern initially seem weaker candidates for evaluating the presence of CCCs, although some have been already identified (Figure 1.1b).

Table 4.2: Relation between tokens of distinct concerns [15]

| Tokens | PMI | Co-occurrences |
|---|---|---|
| matlabpool - spmd | 16 | 6 |
| error - nargchk | 8 | 1005 |
| zeros - size | 4.8 | 4106 |
| ones - size | 4.7 | 2053 |
| nargin - error | 4.7 | 1725 |
| ones - length | 3.9 | 1316 |
| zeros - length | 3.4 | 1944 |

Table 4.3: Relation between tokens and reserved words [15]

| Token and keywords | PMI | Co-occurrences |
|---|---|---|
| otherwise - error | 7.1 | 2488 |
| elseif - isnumeric | 5.6 | 415 |
| disp - try | 5.4 | 1001 |
| catch - disp | 5.3 | 875 |
| elseif - iscell | 5.2 | 273 |
| elseif - isa | 5.0 | 343 |
| elseif - isstruct | 5.0 | 208 |
| elseif - ischar | 4.9 | 427 |
| if - nargin | 4.9 | 15196 |

Table 4.2 presents the pairs of concern tokens from distinct concerns that presented higher PMI value. The higher PMI value the more frequent a concern token is accompanied by the other concern token. The first row (corresponding to the pair of concern tokens 'matlabpool' and 'spmd') shows too few co-occurrences in the file repository, but when they occur it is usually in compliance. Although concern tokens 'error' and 'nargcheck' demonstrate lower PMI, they occur more times together, deeming a strong candidate for studying aspectual features.

### 4.2.5 Schizophrenic functions

As previously demonstrated, a practice familiar to MATLAB users is to create functions with (un)plugable functionalities, sometimes with several functionalities. In Figure 4.1 is one example of a function that its additional functionality depends on a variable (p) treated as boolean to toggle on or off the additional functionality, which is the data plotting. This function has no knowledge on how it is going to behave until it is called, a behaviour referred in this thesis as *schizophrenic*.

*Schizophrenic functions* are functions that hold several behaviours in their implementation, but the execution flow varies according to the arguments it is called with. Studying and observing this behaviour showed that the use of conditional statements, such as 'if' and 'switch' statements, are essential to verify the function's arguments and define the function's flow. Listing 4.1 presents another example of a schizophrenic function that includes several CCCs. The 'feedback' input variable is responsible for toggling some form of feedback/messages to the user and its value depends on the number of arguments the function is called with.

Listing 4.1: Schizophrenic function example [31]

```matlab
functionE0 = gaborconvolve(im, nscale, norient, minWaveLength, mult,
    sigmaOnf, dThetaOnSigma , feedback )
if nargin == 7
  feedback = 0;
end
... % original code removed
if ~isa(im,'double')
  im = double(im);
end
... % original code removed
clear x; clear y; clear theta; % save a little memory
... % original code removed
for o = 1:norient, % For each orientation.
  if feedback
    fprintf('Processing orientation %d \r', o);
  end
  ... % original code removed
end
if feedback, fprintf('     \r'); end
```

Several function examples presenting the use of conditional statements concurrently with 'Verification of function arguments and return values' functions (entry 1. in Table 4.1) (mainly the 'nargin' function, which returns the function's number of input arguments) confirmed the relation between these patterns and the schizophrenic behaviour (Listing I.6). Jota [15] validated this practice and use of such patterns during his study, where he observed a relevant number of occurrences of the token pattern 'if + nargin' (Table 4.3).

Functions such as the plotting example (Figure 4.1) do not contain any token-based information that can be considered consistent throughout MATLAB systems, except for the token 'plot' (entry 7. in Table 4.1). The pattern 'if + nargin' is consistent in MATLAB systems and a more accurate indicator of the presence of schizophrenic functions, which receive further attention in this thesis (Section 6.3).

### 4.2.6 Intelligent MATLAB code repository

The intelligent repository was developed by Relvas et al. [33, 34] with the purpose of accommodating in one place all the data needed to perform exploratory analyses on the MATLAB code base, by means of advanced analysis components. Figure 4.2 shows the entity-relation diagram (ER diagram) that partially makes the structure of the intelligent MATLAB repository. Searching token patterns in the code repository tended to become increasingly elaborate and structured, since we are always interested in finding new patterns, perform studies and check results. This opens the door to wider metric extraction from the code base [33].

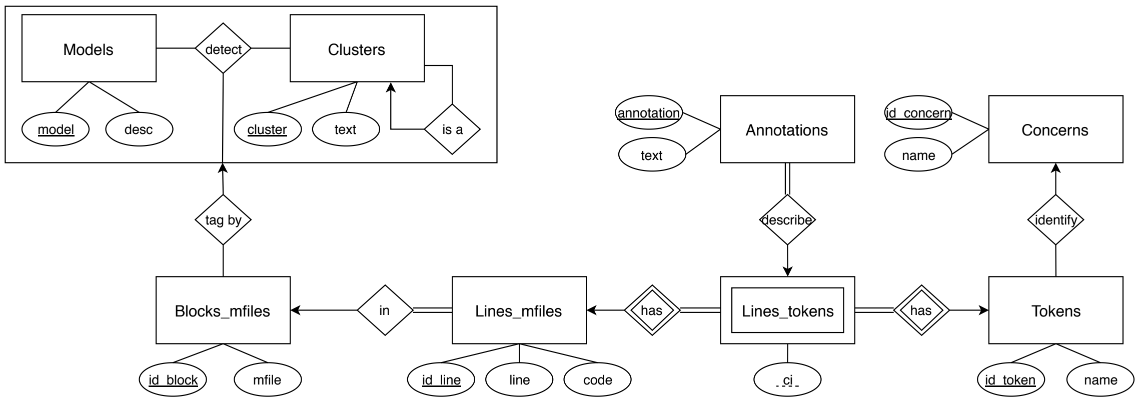


Figure 4.2: Intelligent repository systems ER diagram for token related entities [34]

The repository management system supports intelligent queries over code files associating them to higher level concepts, supported by a web interface. This is achieved by the synergistic combination of a token extraction system, a relational database and the advanced exploratory capabilities of a Self-Organizing Map (SOM) [19]. The database is built on top of a variation of the SOM algorithm, the Ubiquitous Self-Organizing Map (UbiSOM) [37], that allows data streaming into the database of future MATLAB files [8].

The part responsible for storing information related to source code and code structuring, which is the most relevant for this thesis, is composed of the following tables:

- Concerns - holds information about the concerns from Table 4.1;
- Tokens - holds information on individual (unique) tokens in the repository and associates tokens with the 'Concerns' table as presented in Table 4.1. Tokens not related to a concern have no association to the 'Concerns' table;
- Blocks_mfiles - holds information about an m-file, treating it as a single block;
- Lines_mfiles - holds information on each line of a an m-file (Blocks_mfiles), its line number and the code string for ease of display in the web interface. 'Blocks_mfiles' and 'Lines_mfiles' are sufficient to re-write each m-file of the repository;

- Lines_tokens - links the token-concern mapping (Tokens and Concerns) to the lines of each m-file (Lines_mfiles), associating a token to a line with the help of its weak key, 'ci', that stands for "coluna inicial"(initial column). It is the most important table that connects the aspect mining technique to the source code and enables the extraction of metrics.

## 4.3 Aspect-Oriented Extensions of MATLAB

Previous studies on aspect mining and AOP focused on MATLAB demonstrated substantial development of aspect-oriented extensions for modularization of CCCs in the procedural language. Aslam et al. [1] and Cardoso et al. [7] contributed with two of the most relevant languages that serve as aspectual extensions to MATLAB.

### 4.3.1 LARA - MATISSE

MATISSE is LARAs weaving engine for MATLAB code and relies on LARA aspects for specifying data types, shapes, and code instrumentation and specialization. Currently, in addition to being an aspect-oriented extension to MATLAB, it supports a subset of MATLAB as its input, and generates MATLAB and C programming language code (Figure 4.3). It uses LARA aspects to guide the application of source-to-source transformations in its internal high-level code representation, as well as variable type and shape definitions when generating C code from a data structure or code used internally by the compiler to represent source code, or intermediate representation (IR) [4, 42].

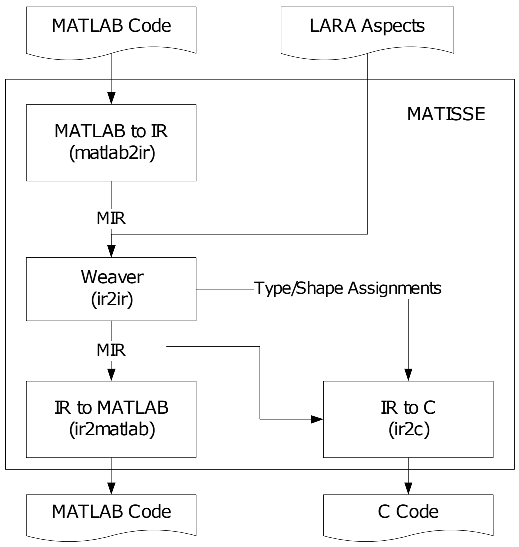


Figure 4.3: The MATISSE compiler framework [4]

LARA is connected to the weaving engine, MATISSE, that is responsible for building the IR, an AST of the code, for the target application, using it to select the points where the aspect is to be inserted and generate the modified code. ASTs can be exploited for aspect mining techniques, as previously mentioned in subsection 3.5.1, making LARA a viable tool for aspect mining in MATLAB code bases.

### 4.3.2 AspectMatlab

Aslam et al. developed AspectMatlab, an AOP language specific to the MATLAB language. In AspectMatlab, aspects are defined using a syntax similar to MATLAB classes, typically containing properties, methods, events and enumerations (Section 2.3). Taking advantage of a MATLAB class structure, an aspect in AspectMatlab retains the properties and methods, while adding two aspect-related constructs: patterns and actions. Patterns specify where to exactly apply the action and of course, actions correspond to the aspect instructions. This choice of terminology was intended to convey that patterns specify where to match and actions specify what to do [1].

## 4.4 Summary

Previous studies adapted and matured a token-based aspect mining approach for MATLAB systems, relating some tokenized function names and reserved words to certain concerns. CCCExplorer is the tool developed and used for tokenizing MATLAB source code and extract token metrics from MATLAB repositories, which enabled such studies.

This chapter presented a study on the MATLAB language from a token-based perspective based on previous studies:

- Token-concern relations - Tokens that maintain consistent meaning throughout several MATLAB systems can be explored for the detection of unmodularized concerns;
- Detection of CCCs based on the co-occurrence of pairs of tokens - Frequent token co-occurrences show points of greater interest and more complex code symptoms, such as the co-occurrence of tokens 'if' and 'nargin' that show a strong relation with schizophrenic functions;
- Development of an intelligent repository for MATLAB code - The need to continuously refine and adapt the existing tools has led other studies to develop better tools for code exploration and metric extraction, such as an intelligent repository for MATLAB code.

# Chapter 5

# System Extension Implementation

*The following chapter describes the motives that guided the extension and implementation of blocks of code into the previously mentioned tools. Then it discusses the system (CCCExplorer) refactoring decisions to enable the implementation of the extension (code blocks) into the system, providing technical information about the system, design decisions and structure.*

## 5.1 Revising CCCExplorer

The initial purpose of the Java written tool, CCCExplorer, was to decompose MATLAB source files into sequences of tokens extracted from the code and allowed the computation of metrics based on those tokens. Relvas [34] used CCCExplorer's tokenization capabilities to create the intelligent repository, enriching CCCExplorer with additional code responsible for generating a series of insert commands that would reflect the MATLAB source file repository in an SQL database format.

As in many instances mentioned by Martin Fowler [10], systems often have to be refactored in order to retain a more convenient structure for adding new features:

> *When a software system is successful, there is always a need to keep enhancing it, to fix problems and add new features. (...) Often enhancements are applied on top of each other in a manner that makes it increasingly harder to make changes. Over time new work slows to a crawl. To combat this change, it is important to refactor code so that added enhancements do not lead to unnecessary complexity* (Fowler 2000).

The code responsible for generating the SQL insert commands was scattered throughout several classes in the system and needed to be enclosed in its own class. Also, not all MATLAB syntax elements studied during this thesis were contemplated as Java objects in the system. Tokens, files and toolboxes were the essential entity objects to fulfill the initial purpose of CCCExplorer, however, the new system requirements require exposing blocks and lines of code as first class entities. The study on the MATLAB language from chapter 2 was an essential part of the refactoring and integration of the block context into the system.

### 5.1.1 Refactoring decision

To refactor CCCExplorer, the system was studied and its behaviour analysed with the help of unitary tests developed in previous studies. The code responsible for generating the SQL inserts was not covered by tests and was hindering the readability of the remaining system's code in a way that led to the decision of restructuring the system by re-implementing the SQL insert generator feature in a previous and more stable version of CCCExplorer, respecting good programming practices. This way CCCExplorer was more easily interpreted, consequently facilitating the addition and isolation of the SQL generator feature, which was included in an 'SQLProducer' class.

### 5.1.2 Block feature addition

Both lines and blocks were classes developed and integrated in CCCExplorer's parsing process. Initially, blocks were thought to be neatly organized and have their respective lines associated, however, statement blocks (Sub-section 2.2.4) have a particular characteristic: a single line of code can hold several statement blocks, as long as they remain correctly delimited. Therefore, blocks are considered aggregations of tokens, as lines are, and other blocks. A block contains tokens (at least its identifier keyword) and may or not contain other blocks.

Lines of code are divided in two types and can be considered MATLAB code or comments. MATLAB block types cluster statement types (Subsection 2.2.4), class related blocks (Section 2.3), functions (Subsection 2.2.5) and, the top-most block, the m-file (Section 2.4).

### 5.1.3 Considerations and documentation

The implementation and refactoring of the new requisites improved CCCExplorer's scalability, maintainability and notion of MATLAB sources. Also, the new system's structure is partially represented by the class diagram in figure 5.1, documentation developed during the process. Now, CCCExplorer can generate a series of SQL insert commands to create an SQL database that better reflects the m-file repository as a pluggable/unpluggable functionality.

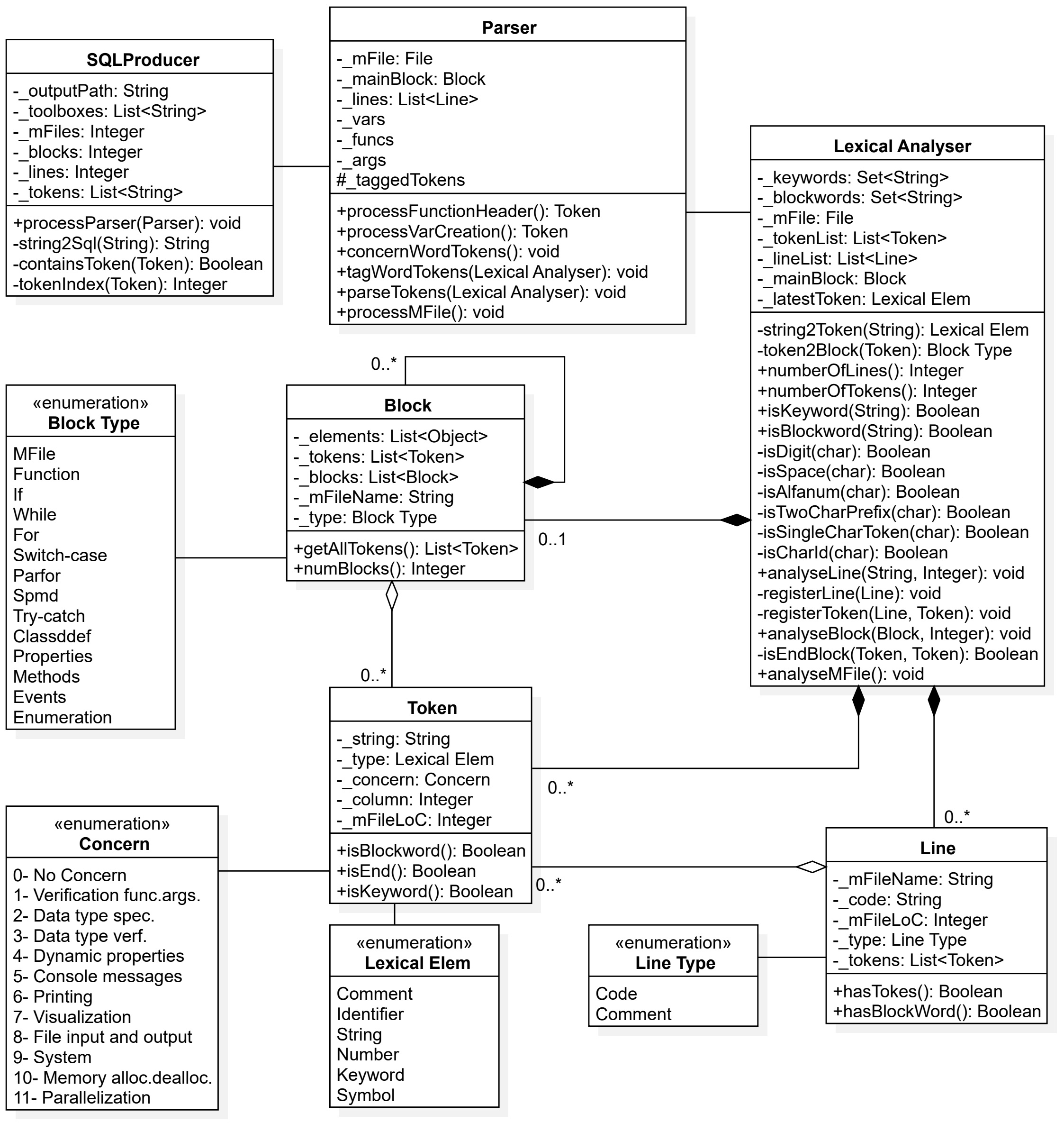


Figure 5.1: Partial structure of the class diagram that represents CCCExplorer

## 5.2 Extending the Intelligent Repository

The refactoring performed on the Java written tool also allowed the extension of the intelligent repository with a context of blocks of code. The extension offers a more accurate reflection of MATLAB source code in the database, refining the notion of MATLAB code blocks.

The position of each individual token in the repository is stored in the 'Lines_tokens' table, which links the token to a line with the help of a weak key that represents the initial column (ci) of the token on its line. Each individual token can be identified with the weak entity 'Lines_tokens' and its weak relations to 'Token' and 'Lines_mfiles', therefore it does not need a key from 'Blocks_mfiles' for that purpose. Figure 5.2 shows the revised

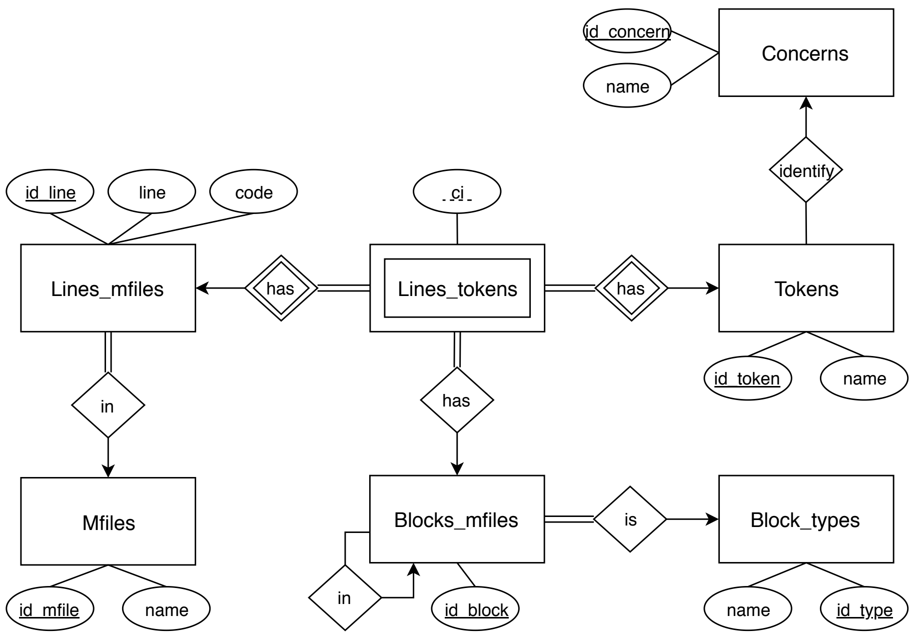


Figure 5.2: Intelligent repository revised

ER diagram that partially makes the structure of the intelligent MATLAB repository.

Two tables were added to maintain consistency and assure the system's continuous evolution. Table 'Block_types' holds the information about the different types of blocks in MATLAB (classdef, properties, methods, events, enumeration, function, if, while, for, switch, parfor, try, spmd), including the block type related to the whole m-file. Table 'Mfiles' acts as 'Blocks_mfiles' from the diagram in Figure 4.2, except now the block information representing the whole m-file (and other blocks) is stored in 'Blocks_mfiles' and associated to the m-file information through the 'Lines_mfiles' table.

## 5.3 Summary

CCCExplorer, the token-based aspect mining system, was refactored for the purpose of adding new features. It was extended with a context of code blocks which enables the extraction of a wider range of metrics. Furthermore, the intelligent repository was extended with the same context, offering a more accurate reflection of MATLAB source code to the token-based aspect mining tool.

Chapter 6

# Code Block Analysis and Results

*The following chapter analyses the MATLAB repository from the perspective of blocks of code with the help of the developed extension to the intelligent repository. It presents a brief overview of the repository and, based on the tailored notion of CCC in MATLAB, it follows with an analysis on each type of block and the particular case of schizophrenic functions.*

## 6.1 Repository Overview

As previously mentioned in section 4.2, the m-file repository was expanded to 65 thousand m-files (Subsection 4.2.2), was subjected to the same tokenization process by CCC-Explorer (Subsection 4.2.1) and the suitably converted data inserted into the intelligent repository database (Subsection 4.2.6). This process simplified the tasks of analysing the content of those m-files, where a user may deploy simple queries to obtain an overview on the repository.

| | |
|---|---|
| M-Files | 65.732 |
| Tokens | 46.234.286 |
| Concern Tokens | 996.933 |
| Distinct Tokens | 321.903 |
| Lines | 3.937.807 |
| Blocks | 619.955 |

Table 6.1: Repository composition

Table 6.1 presents the dimension of the repository in its total (Listing II.2). Concern tokens are the total number of occurrences of concern tokens (tokens related to the concerns from table 4.1). Distinct tokens are, as the term implies, the number of distinct tokens in the repository.

### 6.1.1 Blocks

The approximately 620 thousand blocks in the repository (Table 6.1) are distributed amongst the existing MATLAB code block types as presented in table 6.2 (Listing II.3). As a verification measure, the amount of 'm-file' blocks is presented and also corresponds to the amount of m-files in the repository.

| Block type | Occurrences |
|---|---|
| m-file | 65.732 |
| function | 104.303 |
| if | 305.390 |
| while | 8.728 |
| for | 102.342 |
| switch | 13.393 |
| parfor | 644 |
| spmd | 44 |
| try | 9.625 |
| classdef | 1.662 |
| properties | 2.321 |
| methods | 3.103 |
| events | 2.655 |
| enumeration | 13 |

Table 6.2: Block occurrences

Nearly 50% of the blocks are 'if' blocks, with a ratio of 4-5 'if' statement blocks per m-file and almost 3 per function. 'Switch' statement blocks can also control the flow of a program, even in a more structured way when facing several flows in a single function or file. However, as opposed to the amount of 'if' statements, 'switch' statements make just 2% of the total blocks in the repository. These values are a promising indicator of bad coding practices.

About 2,5% of of the blocks are 'classdef' blocks, but since a single class file contains a single 'classdef' block, approximately 15% of the m-files are class files. This is a strong indicator of the emerging MATLAB user's need to develop more reliable and maintainable MATLAB systems, as opposed to the scripting and fast prototyping features MATLAB mainly offers.

Only 1% of the blocks belong to the Parallel Computing Toolbox ('parfor' and 'spmd'), which is too small an amount to reliably derive conclusions based in token densities and concerns. Acquiring official MATLAB toolboxes has a certain cost associated that most likely justifies the scarce amount of open source code related to such toolboxes.

### 6.1.2 Concern Tokens

Nearly 1 million of all the tokens in the repository are concern tokens (Table 6.1). The distribution of each concern in those 1 million tokens can be observed by their token ratio in the graph presented in figure 6.1.

The graph presents a few different token ratios between concerns, where a little over 30% of the concern tokens are related to the data type verification concern (DTV - entry 3. from table 4.1), and approximately 20% of the concern tokens being related to the visualization concern (Vis - entry 7. from table 4.1). Other concerns make the remaining 50% of the total concern tokens, some nearing to 10% and others not even reaching 2% of that total.

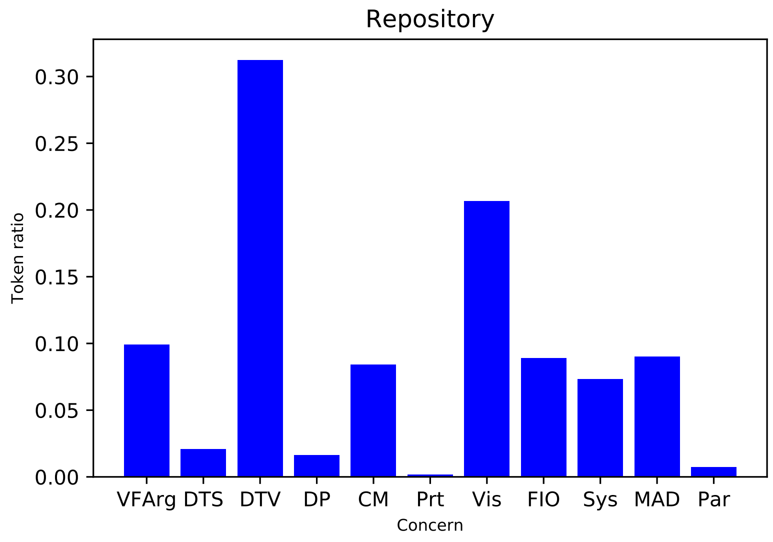


Figure 6.1: Token ratio vs Concern - Repository

Although not approached in this study, these discrepancies are good indicators of the necessity of more in-depth studies on each concern and possibly the refinement of the notion of CCC introduced by Monteiro et al. [8, 31, 32]. This chapter focuses on the general perspective of the presence of these concerns in blocks of code, specifically the detection of concern tendencies in certain types of blocks.

## 6.2 Blocks and Concerns

The study conducted by Jota [15] on the relation between tokens and reserved words reported a few relevant points of interest, which are presented partially in table 4.3. These particular relations in table 4.3 were selected from the study since they contained block-related keywords and could be transposed to the objective of this thesis.

Observing table 4.3, 'otherwise' is a keyword that is reserved for the 'switch' statement, used to indicate the flow of the program if none of the other conditions in the statement are verified (Listing 2.5). 'Elseif' is a keyword reserved for 'if' statements (Listing 2.3). Finally, 'try' and 'catch' are keywords reserved for the 'try-catch' statement.

As a result, this thesis verified the occurrence of the tokens inside the respective blocks in the expanded repository, presenting the obtained values in table 6.3 (Listing II.4). Symptoms of the presence of schizophrenic functions, i.e. 'if' statement blocks with the 'nargin' token, remain as one the most relevant patterns, one that is focused further in section 6.3.

| **Block + token** | **Co-occurrences** | **Token occurrences** |
|---|---|---|
| if + nargin | 26.928 | 29.312 |
| if + iscell | 3.417 | 3.596 |
| if + isa | 3.916 | 4.437 |
| if + isstruct | 2.463 | 2.708 |
| if + ischar | 4.854 | 5.424 |
| try + disp | 2.161 | 35.705 |
| switch + error | 4.079 | 38.105 |
| switch + disp | 968 | 35.705 |

Table 6.3: Occurrence of a block with a concern token

During the following subsections several density bar graphs are presented. The token ratio (Y axis) of each numbered concern from table 4.1 (X axis) was calculated for particular block types. Each graph presents the concern token ratio metric for three distinct domains:

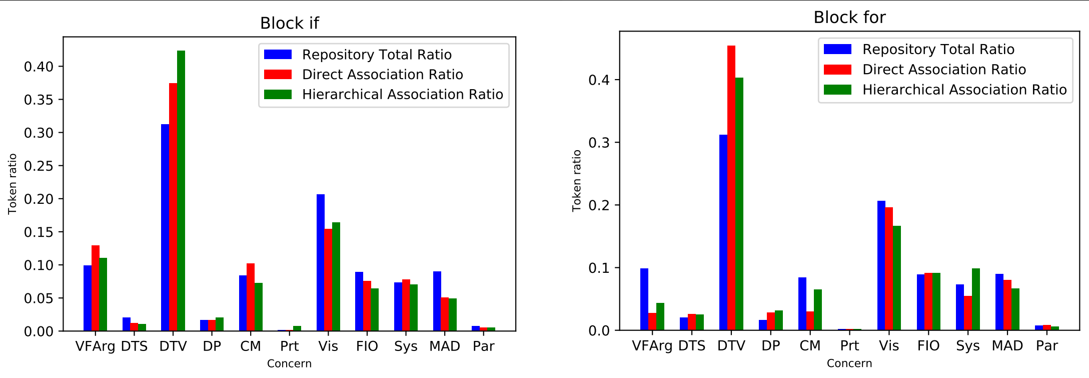
1. the entire repository (blue). This ratio will serve as a reference for the entire repository's concern token ratio and is used to facilitate the comparison between the overall concern tendency of the repository with the type of block under discussion, therefore its value is the same in every graph (as in figure 6.1);

2. tokens directly associated to blocks of the type under discussion (red);

3. tokens associated to sub-blocks of blocks (or hierarchically associated) of the type under discussion (green). Tokens hierarchically associated to a particular block are those directly associated to blocks that are contained in the particular block at any level of containment, thus the term 'hierarchical'.

### 6.2.1 For and If Statement Blocks

Statement blocks 'if' and 'for' are the most common in the repository and present concern token ratios similar to the repository, which does not expose any clear tendency from using these types of blocks. However, when compared to other types of blocks, they are the ones of utmost importance that present concern token ratios slightly above the repository's ratio regarding the DTV concern (Figure 6.2).



Directly associated concern tokens = 466.355
Hierarchically associated concern tokens = 584.057

Directly associated concern tokens = 124.800
Hierarchically associated concern tokens = 206.603

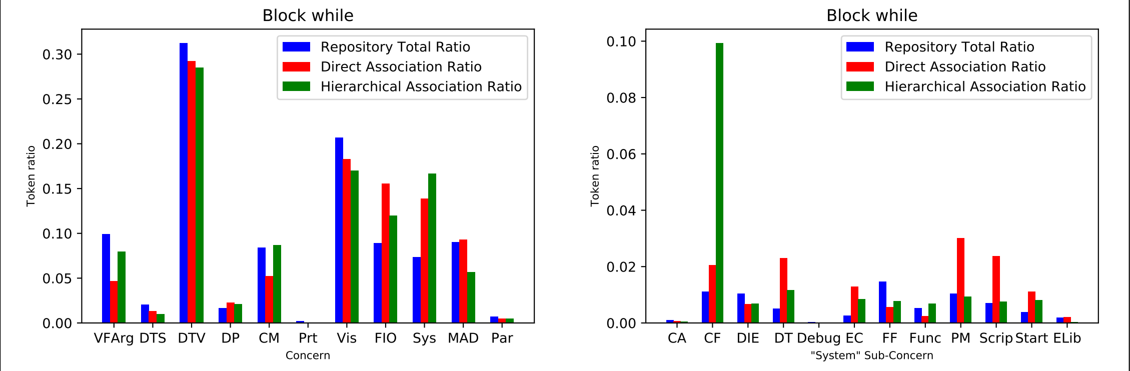
Figure 6.2: Token ratio vs Concern - if and for blocks

Together with the metrics in table 6.3, this pattern ('for', 'if' and DTV) is a strong evidence supporting a behaviour where both statements deal with function input arrays storing different data types (Subsection 2.2.2). Not only these statement blocks make the majority of blocks in the repository, this 'data type schizophrenia' behaviour is present in 6.995 out of the 65.732 m-files, corresponding to over 10% of the files in the repository (Listing II.5). These are relevant observations that support the use of these types of patterns of statements and tokens to implement the mentioned behaviour.

Two random MATLAB file examples presenting the stated behaviour were extracted from the repository (Listings I.2 and I.3). In both, a 'for' statement block is used for iterating the function's input array and the program will behave differently according to what type of data is stored on each cell of the array, using 'if' statements to verify the data type of the cell and implement the function's desired flow for that data type.

### 6.2.2 While Statement Block

Figure 6.3 presents the concern token ratio values extracted from 'while' statement blocks in the repository. The graph presents ratios rather similar to the repository's, with the exception of the system concern (Sys - entry 9. from table 4.1). Among all statement block types, the use of this statement is the one that presents the highest ratio of tokens related to this concern. The system concern was analysed more thoroughly and the ratios of its sub-concerns are also presented in figure 6.3.



Directly associated concern tokens = 12.486

Hierarchically associated concern tokens = 40.371

Figure 6.3: Token ratio vs Concern - while blocks

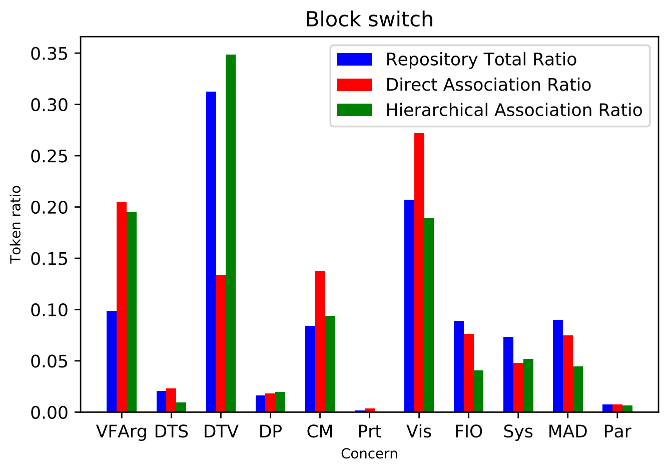
The higher ratio of tokens hierarchically associated to 'while' blocks regarding the control flow sub-concern (Sys-CF) is a promising indicator of a consistent behaviour in 'while' statement blocks. The amount of m-files in the repository that have Sys-CF concern tokens inside blocks hierarchically associated to 'while' blocks is equal to 1.699 m-files (Listing II.6) from which two random MATLAB file examples were extracted from the repository (Listings I.4 and I.5). Both examples present a pattern where Sys-CF concern tokens appear inside 'if' statement blocks, and these 'if' statement blocks are inside the 'while' statement block.

This pattern reflects a behaviour where these types of tokens are used to control 'while' statement iterations and the program's flow. Moreover, token 'continue' is also present and used in the code as a function with the purpose of controlling loops and loop iterations [21], one that could also be inserted into the concern-token relation table's Sys-CF sub-concern (Table 4.1).

### 6.2.3 Switch Statement Block

Figure 6.4 presents the concern token ratio values extracted from 'switch' statement blocks in the repository. The ratio of verification of function arguments concern tokens (VFArg - entry 1. from table 4.1) associated to 'switch' statement blocks is twice as the repository's ratio, which supports the possibility of 'switch' statements also presenting a schizophrenic behaviour.

The ratio of directly associated console messages concern (CM - entry 5. table 4.1) tokens supports the use of 'switch' statement blocks with 'error' token, as presented in table 6.3, which validates the occurrence of the pattern 'otherwise + error' studied by Jota [15], presented in table 4.3.



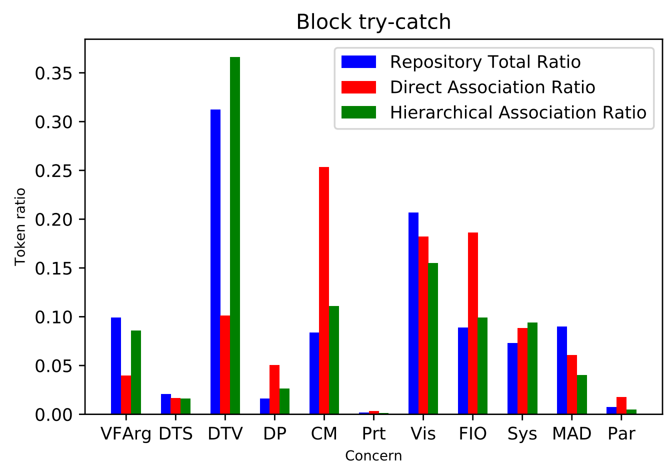
Directly associated concern tokens = 38.617
Hierarchically associated concern tokens = 85.046

Figure 6.4: Token ratio vs Concern - switch blocks

About 1.466 MATLAB files in the repository present 'switch' statement blocks with schizophrenic symptoms (Listing II.7), from which two examples were extracted (Listings I.6 and I.7). Not only these symptoms can be confirmed by the presence of 'nargin' keyword, both examples also present 'switch' statements with the 'error' token after the 'otherwise' keyword of the statement. This corresponds to a behaviour implemented to send customized error messages to the user if none of the other conditions in the statement are met.

### 6.2.4 Try-Catch Statement Block

Figure 6.5 presents the concern token ratio values extracted from 'try-catch' statement blocks in the repository. Compared to the repository, this statement presents a higher ratio of directly associated tokens related to CM and file I/O concerns (FIO - entry 8. respectively from table 4.1).



Directly associated concern tokens = 11.744
Hierarchically associated concern tokens = 24.167

Figure 6.5: Token ratio vs Concern - try-catch blocks

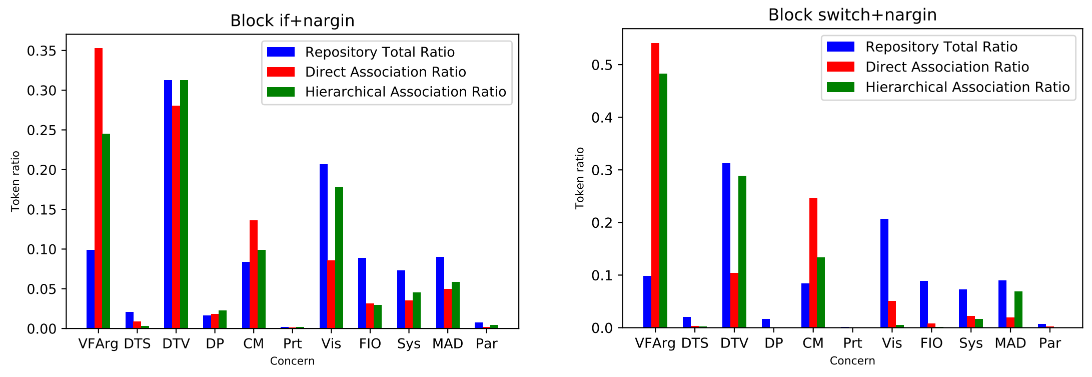
There are 940 m-files presenting 'try-catch' statement blocks with CM concern tokens. Example I.9 supports the use of these statements with the 'disp' token, as noted in table 6.3. Furthermore, it was also observed the use of the 'error' token for the same purpose, as presented in example I.8. This pattern refers to the customization of error messages if MATLAB encounters an error while executing the code after the 'try' keyword and before the 'catch' keyword.

'Try-catch' statements are also commonly used for reading files and, in case of failure due to missing input files, implement an alternative. A total of 781 m-files present a pattern with 'try-catch' statements including FIO concern tokens. Observed alternative behaviours include simply sending error messages to the user (Example I.10), initialize variables (Example I.10) or read other files that the programmer estimates they exist unconditionally.

## 6.3 Schizophrenic Functions

As in the previous section, the following density bar graphs present three density metrics: the concern token ratio from the entire repository (blue); the concern token ratio of tokens directly associated to the block under discussion (red); the concern token ratio of tokens hierarchically associated to the block under discussion (green).

In figure 6.6 is presented the token ratio of each concern in blocks that present the schizophrenic symptom, identified by the ’if’ statement block (on the left) and ’switch’ statement block (on the right) containing the ’nargin’ token directly associated to it. For each statement, one occurrence of ’nargin’ token per statement block was excluded from the ratio to better study behaviours beyond the main one, the schizophrenia.



Directly associated concern tokens = 25.100
Hierarchically associated concern tokens = 59.817

Directly associated concern tokens = 1.997
Hierarchically associated concern tokens = 13.712

Figure 6.6: Token ratio vs Concern - if and for blocks

There are 12.962 m-files presenting ’if’ statement blocks containing the ’nargin’ token and 500 m-files presenting ’switch’ statement blocks containing the ’nargin’ token in the repository (Listings II.10 and II.13). In other words, about 20% of the files in the repository present function schizophrenia symptoms, validating the same observations made by Jota [15].

Even though one occurrence of ’nargin’ was excluded from the calculated ratios, the VFArg concern token ratio presents a much larger value when compared to regular ’if’ and ’switch’ statement blocks. This is an indicator of functions that present schizophrenic symptoms have a decreased ability to understand their reality and plausibly holding multiple (more than two) behaviours on which they may be called.

## 6.4 Summary

The m-file repository was tokenized and analyzed from a code block perspective. Observations show that the repository has too few blocks of certain types and more m-files should be added for future studies. Also, some block types are use more frequently than others and some patterns of usage of such blocks present consistent behaviours throughout the repository.

The token ratios observed inside function files with schizophrenic symptoms indicate that these functions tend to have multiple behaviours implemented. These tangled behaviours are promising signs of bad programming practices and/or limitations in the MATLAB language. The use of the patter consisted of 'for' and 'if' statement blocks with data type verification MATLAB functions relates to the code necessary to deal with the possibility of storing different data types in the same array. 'Try-catch' and 'switch' statement blocks are strongly related to the implementation of customized error messages and feedback messages.

# Chapter 7

# Conclusions and Future Work

*This last chapter presents a summary of the work conducted during this thesis, including results obtained and opportunities for future work.*

## 7.1 Summary

As software systems grow in size and complexity a need for developing tools to help in management and further develop those systems also grows. Problems regarding concern detection and modularization will continue to exist up to some level or granularity in programming languages. MATLABs components limit in some way the structure of a system, having a negative impact on maintainability, reliability and performance. The need to study its syntax comes apace with its limitations, barging the door to studying and adapt already existing methods for, ultimately, solving some of these issues. Symptoms of the presence of unmodularized concerns, or CCCs, persist in existing at some level in software systems regardless of how a system is divided in modules. Therefore, more effort should be aimed towards continuously develop, research and refine CCC detection techniques.

This thesis studied a token-based approach for detecting concerns tailored specifically for MATLAB systems. CCCExplorer is a tool used for tokenizing MATLAB source files and compute metrics based on the tokens generated, recently upgraded to an intelligent repository for source files. Consequently, it increased the degree to which MATLAB source code can be analysed. This research analysed the intelligent repository, revealing imprecise reflection of MATLAB code sources by not contemplating its 'code blocks'. A refactor was made to CCCExplorer, enhancing it with a notion of MATLAB block of code, represented by the diagram in Figure 5.1, and implementing a code block context in the intelligent repository, represented by Listing II.1.

The study on MATLAB code blocks contributed with a better analysis of its syntax and structure chapter 2. It confirmed that language offered more possibilities beyond the initially implemented structures and contexts, noting some of the to-be-made contributions to the token-based approach. Results extracted during this thesis have confirmed that the implemented MATLAB code block context is a promising addition to the tool and an improvement to CCCExplorer and the token-based approach.

Results regarding concern-related token metrics and blocks show that particular blocks tend to have some type of relation with certain concerns. Some blocks were excluded from the observations due to low number of occurrences, particularly the ’parfor’ and ’spmd’ parallel statement blocks.

Considering the detected patterns, the most significant one is composed of ’for’ and ’if’ statement blocks with data type verification MATLAB functions. It is used for implementing several behaviours in a single function and its flow is different according to the type of data in each cell of the function’s input array. As schizophrenic functions, the discovered patterns are strong evidence supporting the existence of modularity limitations in the MATLAB language and should be thoroughly analysed in future studies.

Schizophrenic functions were targets of interest to the extraction of token and block metrics. Promising results showed that ’if’ statement blocks with the ’nargin’ token tend to have even more tokens related to verification of function arguments and return values (entry 1. from Table 4.1), possibly increasing their level of schizophrenic behaviour. ’Switch’ statements with the ’nargin’ token present similar behaviour, although at an inferior level, however, they also show a relevant presence of tokens related to console messages, possibly related to the implementation of different behaviours for sending feedback or error messages to the system user.

## 7.2 Future Work

Separation of concerns in MATLAB systems is still considered a path to follow in subsequent studies, regardless of the information gathered. One of the main reasons is that the current m-file repository is modest and lacks representativeness of MATLAB systems in a global perspective.

The python scripts presented in this thesis can be used to further increase the m-file repository, so that more MATLAB source code can be studied and consequently improve CCCExplorer with more precise structures to represent the code. This will favor further studies focused on refining the notion of concern initially developed by Monteiro et al. or perhaps rethink it. However, an implementation of cloned file detectors is needed to avoid repetitive information and maintain and unbiased repository.

CCCExplorer was studied and refactored in a way that it would be easier to include new features and functionalities to the tool. It has grown to a relevant degree of complexity and it is expected that further studies will continue to use it, which will require spending some effort in its comprehension. It is also imperative to develop documentation and improve CCCExplorer's readability and maintainability, as well as keeping CCCExplorer a well modularized tool.

This study also presented an analysis on object-oriented programming in MATLAB that can be relevant towards the CCC matter. Documented refactoring techniques take in consideration the use of objects to modularize systems in a series of ways, leaving room for exploration and, ideally, application towards solving some of the patterns and CCCs studied. A relevant example supporting their application would be transforming a schizophrenic function into a class, separating each of its modes into different functions.

The token-based approach still needs refinement, but other techniques can still be study subjects to concern mining in MATLAB sources, such as ASTs. The AOP language LARA handles MATLAB source code using ASTs and can be used as a tool for incorporate future approaches to concern detection. However, the language is still scarcely documented and preliminary analysis has shown difficulties on manipulating a large number of m-files.

# Bibliography

[1] T. Aslam, J. Doherty, A. Dubrau, and L. Hendren. "AspectMatlab: An aspect-oriented scientific programming language." Doctoral dissertation. McGill University, 2010.

[2] Baeldung. *Intro to AspectJ*. https://www.baeldung.com/aspectj. [Online; accessed 21-February-2019].

[3] C. Y. Baldwin and K. B. Clark. *Design rules: The power of modularity*. Vol. 1. MIT press, 2000.

[4] J. Bispo, P. Pinto, R. Nobre, T. Carvalho, J. M. Cardoso, and P. C. Diniz. "The MATISSE MATLAB compiler." In: *Industrial Informatics (INDIN), 2013 11th IEEE International Conference on*. IEEE. 2013, pp. 602–608.

[5] M. Bruntink, A. Van Deursen, T. Tourwe, and R. van Engelen. "An evaluation of clone detection techniques for crosscutting concerns." In: *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*. IEEE. 2004, pp. 200–209.

[6] J. M. Cardoso, J. M. Fernandes, and M. P. Monteiro. "Adding aspect-oriented features to matlab." In: *Fifth International Conference on Aspect-Oriented Software Development (AOSD 2016)*. 2006.

[7] J. M. Cardoso, T. Carvalho, J. G. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov. "LARA: an aspect-oriented programming language for embedded systems." In: *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. ACM. 2012, pp. 179–190.

[8] N. Cavalheiro Marques, M. Monteiro, and B. Silva. "Analysis of a token density metric for concern detection in Matlab sources using UbiSOM." In: *Expert Systems* 35.4 (2018), e12306.

[9] M. Cleve Moler. *A Brief History of MATLAB*. https://www.mathworks.com/company/newsletters/articles/a-brief-history-of-matlab.html. [Online; accessed 30-July-2019].

[10] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[11] S. Gaiarin. *doxymatlab*. commit: 9aa1ab07d07c84950db7f5254423864afb74804. 2017. URL: https://github.com/simgunz/doxymatlab.

[12] I. GitHub. *GitHub*. https://github.com/. Accessed: 2019-08-31.

[13] L. Hendren. "Typing aspects for MATLAB." In: *Proceedings of the sixth annual workshop on Domain-specific aspect languages*. ACM. 2011, pp. 13–18.

[14] R. Hilliard. "Aspects, concerns, subjects, views." In: *First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems (at OOPSLA'99)*. Citeseer. 1999, p. 59.

[15] B. Jota. "Métodos para o tratamento de tokens na identificação de concerns em código MATLAB." Master's thesis. Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, 2019.

[16] A. Kellens, K. Mens, and P. Tonella. "A survey of automated code-level aspect mining techniques." In: *Transactions on aspect-oriented software development IV*. Springer, 2007, pp. 143–162.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. "Aspect-oriented programming." In: *European conference on object-oriented programming*. Springer. 1997, pp. 220–242.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. "Getting started with AspectJ." In: *Communications of the ACM* 44.10 (2001), pp. 59–65.

[19] T. Kohonen. "The self-organizing map." In: *Proceedings of the IEEE* 78.9 (1990), pp. 1464–1480.

[20] MathWorks. *Advanced Software Development - MATLAB*. https://www.mathworks.com/help/matlab/software-development.html. [Online; accessed 21-February-2019].

[21] MathWorks. *Documentation*. https://www.mathworks.com/help/matlab/ref/continue.html. [Online; accessed 20-September-2019].

[22] MathWorks. *Error Handling - MATLAB*. https://www.mathworks.com/help/matlab/error-handling.html. [Online; accessed 19-July-2019].

[23] MathWorks. *Getting started with MATLAB*. https://www.mathworks.com/help/matlab/getting-started-with-matlab.html. [Online; accessed 21-February-2019].

[24] MathWorks. *Language Fundamentals - MATLAB*. https://www.mathworks.com/help/matlab/language-fundamentals.html. [Online; accessed 21-February-2019].

[25] MathWorks. *Loops and Conditional Statements - MATLAB*. https://www.mathworks.com/help/matlab/control-flow.html. [Online; accessed 21-February-2019].

[26] MathWorks. *MATLAB Central*. https://www.mathworks.com/matlabcentral/. [Online; accessed 21-February-2019].

[27] MathWorks. *Object-Oriented Programming in MATLAB*. https://www.mathworks.com/discovery/object-oriented-programming.html. [Online; accessed 01-August-2019].

[28] MathWorks. *Parallel Computing Toolbox - MATLAB*. https://www.mathworks.com/help/parallel-computing. [Online; accessed 19-July-2019].

[29] MathWorks. *Programming Scripts and Functions - MATLAB*. https://www.mathworks.com/help/matlab/programming-and-data-types.html. [Online; accessed 21-February-2019].

[30] S. Media. *SourceForge*. https://sourceforge.net/. Accessed: 2019-08-31.

[31] M Monteiro, J Cardoso, and S. Posea. "Identification and characterization of cross-cutting concerns in MATLAB systems." In: *Conference on Compilers, Programming Languages, Related Technologies and Applications (CoRTA 2010), Braga, Portugal*. Citeseer. 2010, pp. 9–10.

[32] M. P. Monteiro, N. C. Marques, B. Silva, B. Palma, and J. Cardoso. "Toward a Token-Based Approach to Concern Detection in MATLAB Sources." In: *Portuguese Conference on Artificial Intelligence*. Springer. 2017, pp. 573–584.

[33] A. Relvas. "Uma Interface Web para Comparação de Métricas Utilizando Mapas Auto-Organizados." Master's thesis. Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, 2019.

[34] A. Relvas, N. Marques, M. Monteiro, and G. Carneiro. "An intelligent repository for MATLAB code." unpublished, submitted to the 2019 edition of the Mining Software Repositories (MSR) conference. 2019.

[35] R. Sanchez and J. T. Mahoney. "Modularity, flexibility, and knowledge management in product and organization design." In: *Strategic management journal* 17.S2 (1996), pp. 63–76.

[36] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K Vijay-Shanker. "Using natural language program analysis to locate and understand action-oriented concerns." In: *Proceedings of the 6th international conference on Aspect-oriented software development*. ACM. 2007, pp. 212–224.

[37] B. Silva and N. C. Marques. "The ubiquitous self-organizing map for non-stationary data streams." In: *Journal of Big Data* 2.1 (2015), p. 27.

[38] O. Spinczyk, D. Lohmann, and M. Urban. "AspectC++: an AOP Extension for C++." In: *Software Developer's Journal* 5.68-76 (2005).

[39] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. "N degrees of separation: multi-dimensional separation of concerns." In: *Software Engineering, 1999. Proceedings of the 1999 International Conference on*. IEEE. 1999, pp. 107–119.

[40] A. Van Deursen, M. Marin, and L. Moonen. “Aspect mining and refactoring.” In: *Proceedings of the 1st International Workshop on Refactoring: Achievements, Challenges, Effects (REFACE), with WCRE*. 2003, pp. 11–21.

[41] Wikipedia contributors. *Concern (computer science) – Wikipedia, The Free Encyclopedia*. [Online; accessed 24-January-2019]. 2018. URL: https://en.wikipedia.org/w/index.php?title=Concern_(computer_science)&oldid=867665311.

[42] Wikipedia contributors. *Intermediate representation — Wikipedia, The Free Encyclopedia*. [Online; accessed 21-February-2019]. 2018. URL: https://en.wikipedia.org/w/index.php?title=Intermediate_representation&oldid=875112602.

[43] Wikipedia contributors. *Cross-cutting concern — Wikipedia, The Free Encyclopedia*. [Online; accessed 21-August-2019]. 2019. URL: https://en.wikipedia.org/w/index.php?title=Cross-cutting_concern&oldid=905886088.

[44] Wikipedia contributors. *Pointwise mutual information — Wikipedia, The Free Encyclopedia*. [Online; accessed 29-August-2019]. 2019. URL: https://en.wikipedia.org/w/index.php?title=Pointwise_mutual_information&oldid=892384067.

Annex I

# MATLAB Code Examples

Listing I.1: Class Definition Example [11]

```matlab
%> @file classdefExample.m
%> @brief File used to show an example of class definition
% ======================================================================
%> @brief Here we have a brief description of the class.
%
%> And here we can put some more detailed informations about the class.
% ======================================================================
classdef  (InferiorClasses = {?class1,?class2}) classdefExample

  properties (Access = protected)
    %> Description of a protected property
    protectedProperty
  end
  properties (Access = public)
    %> Description of a public property
    publicProperty
  end
  properties (Access = private)
    %> Description of a private property
    pivateProperty
  end
  properties (Constant = true)
    %> Description of a constant property
    constantProperty = {'1', '2', ...
        'trois'};
  end
  properties
    %> Description of the first property of the class
    first_property = []
    %> Description of the second property of the class
    second_property = []
    %> Description of the third property of the class
    third_property = [1  2];
  end
  events
    %> Description of first event
    FirstEvent
    %> Description of second event
    SecondEvent
  end
  %> Description of the enumeration.
  enumeration
    %> Description of the first item
    one (1)
    %> Description of the second item
    two (2)
    %> Description of the third item
    three
  end
  methods
    % ======================================================================
    %> @brief Class constructor
    %>
    %> More detailed description of what the constructor does.
    %>
    %> @param param1 Description of first parameter
    %> @param anotherParam Description of the second parametere
    %>
    %> @return instance of the classdefExample class.
    % ======================================================================
    function obj = classdefExample(param1, anotherParam)
    end

    % ======================================================================
    %> @brief Brief description of the exampleMethod1 method
    %>
    %> @param obj instance of the classdefExample class.
    % ======================================================================
    function exampleMethod1(obj)
    end

    % ======================================================================
    %> @brief Brief description of the exampleMethod2 method
    %>
    %> @param obj instance of the classdefExample class.
    %> @retval ret return value of this method
    % ======================================================================
    function ret = exampleMethod2(obj)
    end
  end
  methods (Static=true)
    % ======================================================================
    %> @brief Brief description of the exampleStaticMethod method
    %>
    %> This method is static and public, with an inused (~) argument
    %> @param param1 Description of the parameter
    %> @param param2 Description of the parameter
    %> @retval out return value of this method
    % ======================================================================
    function out = exampleStaticMethod(param1, ~, param2)
    end
  end
  methods (Static, Access=private)
    % ======================================================================
    %> @brief Brief description of the exampleStaticPrivateMethod method
    %>
    %> This method is static and private
    %> @param param1 Description of the parameter
    %> @param param2 Description of the parameter
    %> @retval out return value of this method
    % ======================================================================
    function out = exampleStaticPrivateMethod(param1, param2)
    end
  end
  methods (Access=protected, Static)
    % ======================================================================
    %> @brief Brief description of the exampleStaticProtectedMethod method
    %>
    %> This method is static and protected
    %> @param param1 Description of the parameter
    %> @retval out return value of this method
    % ======================================================================
    function out = exampleStaticProtectedMethod(param1)
    end
  end
  methods (Access=public, Static = true)
    % ======================================================================
    %> @brief Brief description of the exampleStaticPublicMethod method
    %>
    %> This method is static and public
    %> @param param1 Description of the parameter
    %> @retval out return value of this method
    % ======================================================================
    function out = exampleStaticPublicMethod(param1)
    end
  end
  methods (Access=private)
    % ======================================================================
    %> @brief Brief description of the examplePrivateMethod method
    %>
    %> This method is private
    %> @param param1 Description of the parameter
    %> @retval out return value of this method
    % ======================================================================
    function out = examplePrivateMethod(param1)
    end
  end
  methods (Access=protected)
    % ======================================================================
    %> @brief Brief description of the exampleProtectedMethod method
    %>
    %> This method is protected
    %> @param param1 Description of the parameter
    %> @retval out return value of this method
    % ======================================================================
    function out = exampleProtectedMethod(param1)
    end
  end
  methods (Access=public, Static=false)
    % ======================================================================
    %> @brief Brief description of the examplePublicMethod2 method
    %>
    %> This method is public and not static
    %> @param param1 Description of the parameter
    %> @retval out return value of this method
    % ======================================================================
    function out = examplePublicMethod2(param1)
    end
  end
  methods (Access=public, ~Static)
    % ======================================================================
    %> @brief Brief description of the exampleNonStaticPublicMethod3 method
    %>
    %> This method is public and not static
    %> @param param1 Description of the parameter
    %> @retval out return value of this method
    % ======================================================================
    function out = exampleNonStaticPublicMethod3(param1)
    end
  end
  methods (Abstract = true)
    % ======================================================================
    %> @brief Brief description of the exampleAbstractMethod method
    %>
    %> This method is abstract : only the signature of this function is
    %> declared.
    %> @param param1 Description of the first parameter
    %> @param param2 Description of the second parameter
    %> @retval out return value of this method
```

```matlab
    % ================================================================
    out = exampleAbstractMethod(param1, ...
        param2);
end
```

Listing I.2: Example 1 - for if and data type verification

```
function res = struct2xml(s)
  res = [];
  names = fieldnames(s);
  nl_char = sprintf("\n");
  for i = 1:length(names)
    if isempty(s.(names{i}))
      continue;
    end
    if isnumeric(s.(names{i}))
      if length(s.(names{i})) > 1
      else
        res = [res tagged_string( num2str(s.(names{i})), names{i}) nl_char];
      end
    elseif ischar(s.(names{i}))
      res = [res tagged_string(s.(names{i}), names{i}) nl_char];
    elseif isstruct(s.(names{i}))
      for j = 1:length(s.(names{i}))
        res = [res tagged_string(struct2xml(s.(names{i})(j)), names{i}),
            nl_char];
      end
    else
      error("unsupported field type")
    end
  end
end
function res = tagged_string(str, tag)
  res = ["<" tag ">" str "</" tag ">"];
end
```

Listing I.3: Example 2 - for if and data type verification

```
function [olib] = add_struct(ilib, varargin);
    if nargin < 2
       error("gds_library.add_struct :  must have at least two arguments.");
    end
    olib = ilib;
    for k=1:length(varargin)
       S = varargin{k};
       if isa(S, "gds_structure")
          olib.st{end+1} = S;
       elseif iscell(S)
          if ~all(cellfun(@(x)isa(x,"gds_structure"), S))
             error("gds_library.add_struct :  input cell array member is not a
                  gds_structure.");
          end
          olib.st = [ilib.st, S(:)"]; % make row vector
       else
          error("gds_library.add_struct : arguments must be gds_structures or
              cell arrays.");
       end
    end
end
```

Listing I.4: Example 1 - while and control flow

```
function [obj, idx] = object(json, idx, tokens)
    start = idx;
    obj = containers.Map();
    if json(idx) ~= "{"
        error("JSON:parse:object:nobrace", ...
              ["object must start with "{" (char " num2str(idx) ")"]);
    end
    idx = idx+1;
    idx = next(json, idx);
    if json(idx) ~= "}"
        while 1
            if json(idx) ~= """
                error("JSON:parse:string:noquote", ...
                      ["string must start with " (char " num2str(idx) ")"]);
            end
            [key, idx] = string(json, idx, tokens);
            idx = next(json, idx);
            if json(idx) == ":"
                idx = idx+1;
            else
                error("JSON:parse:object:nocolon", ...
                      ["no ":" after object key in "" json(start:idx-1) ...
                       "" (char " num2str(idx) ")"]);
            end
            idx = next(json, idx);
            [val, idx] = value(json, idx, tokens);
            obj(key) = val;
            idx = next(json, idx);
            if json(idx) == ","
                idx = idx+1;
                idx = next(json, idx);
                continue
            elseif json(idx) == "}"
                break
            else
                error("JSON:parse:object:unknownseparator", ...
                      ["no "," or "}" after entry in "" json(start:idx-1) ...
                       "" (char " num2str(idx) ")"]);
            end
        end
    end
    idx = idx+1;
end
```

Listing I.5: Example 2 - while and control flow

```
function [args, cancelled] = optic_disc_mask(vessel_data, args, prompt)

(...)

while true
    if ~ishandle(h)
        return;
    end
    try
        [x, y] = getline(h);
        set(findobj(h, "type", "axes"), ...
                    "ALimMode", "manual", ...
                    "CLimMode", "manual", ...
                    "DataAspectRatioMode", "manual", ...
                    "Drawmode", "fast", ...
                    "PlotBoxAspectRatioMode", "manual", ...
                    "TickDirMode", "manual", ...
                    "XLimMode", "manual", ...
                    "YLimMode", "manual", ...
                    "ZLimMode", "manual", ...
                    "XTickMode", "manual", ...
                    "YTickMode", "manual", ...
                    "ZTickMode", "manual", ...
                    "XTickLabelMode", "manual", ...
                    "YTickLabelMode", "manual", ...
                    "ZTickLabelMode", "manual");
    catch
        return;
    end
    if numel(x) ~= 2
        button = questdlg(["Marking the optic disc diameter requires exactly 2
             points.", char(10), ...
            "Click the image once to set the first point, double-click to set
               the second."],...
            "Optic disc mask","Try again","Cancel","Try again");
        if strcmp(button, "Cancel")
            return;
        else
            continue;
        end
    else
        break;
    end
end

(...)

if args.apply_mask
    set_mask = true;
    vessel_data.optic_disc_mask = [args.min_discs, args.max_discs];
elseif  prompt
    while true
        answer = inputdlg({"Minimum number of optic disc diameters (must be >=
             0)", ...
                    "Maximum number of optic disc diameters (must be larger
                        than minimum)"}, ...
                    "Optic disc diameter mask", ...
                    1, ...
                    {num2str(args_default.min_discs), ...
                     num2str(args_default.max_discs)});
        if isempty(answer)
            break;
        end
        a1 = str2double(answer{1});
        a2 = str2double(answer{2});
        if a1 > 0 && a1 < a2
            args.min_discs = a1;
            args.max_discs = a2;
            set_mask = true;
            vessel_data.optic_disc_mask = [args.min_discs, args.max_discs];
            break;
        end
    end
end

(...)
```

Listing I.6: Example 1 - switch and verification of function arguments

```
function result = getLonAxis(mGridCoordinatesObj)
result = [];
if nargin < 1, help(mfilename), return, end
    try
        switch nargin
            case 1
                result=squeeze(mGridCoordinatesObj.myCoordID.getLonAxis());
            otherwise, error("MATLAB:mGridCoordinates:getLonAxis:Nargin",...
                                    "Incorrect number of arguments");
        end
    catch %gets the last error generated
        err = lasterror();
        disp(err.message);
   end
end
```

Listing I.7: Example 2 - switch and verification of function arguments

```
function [valid, message] = isSBML_FBC_FluxBound(varargin)
switch (nargin)
  case 4
    SBMLStructure = varargin{1};
    level = varargin{2};
    version = varargin{3};
    pkgVersion = varargin{4};
  case 3
    SBMLStructure = varargin{1};
    level = varargin{2};
    version = varargin{3};
    pkgVersion = 1;
  case 2
    SBMLStructure = varargin{1};
    level = varargin{2};
    version = 1;
    pkgVersion = 1;
  case 1
    SBMLStructure = varargin{1};
    level = 3;
    version = 1;
    pkgVersion = 1;
  otherwise
    error("need at least one argument");
end;

(...)
```

Listing I.8: Example 1 - try-catch and console messages

```
function out=nnProc(spd, trq, time, nnFuncs, outputNames, outputUnits)
try
    spd=makeRowArray(spd);
catch
    error("[nnProc.m]: spd array is not correct--please check inputs and try
        again")
end
try
    trq=makeRowArray(trq);
catch
    error("[nnProc.m]: trq array is not correct--please check inputs and try
        again")
end
try
    time=makeRowArray(time);
catch
    error("[nnProc.m]: time array is not correct--please check inputs and try
        again")
end
if length(nnFuncs)~=length(outputNames)&length(nnFuncs)~=length(outputUnits)
    error("[nnProc.m]: outputNames, outputUnits, and nnFuncs must all be of
        the same length")
end
dsdt_05sec=d_dt(spd, time, 5.0); % dspd/dt taken at  5 sec intervals
dsdt_10sec=d_dt(spd, time, 10.); % dspd/dt taken at 10 sec intervals
dTdt_05sec=d_dt(trq, time, 5.0); % dtrq/dt taken at  5 sec intervals
dTdt_10sec=d_dt(trq, time, 10.); % dtrq/dt taken at 10 sec intervals
out=[];
for i=1:length(nnFuncs)
    out=setfield(out, outputNames{i}, feval(nnFuncs(i), [spd; dsdt_05sec;
        dsdt_10sec; trq; dTdt_05sec; dTdt_10sec]));
    out=setfield(out, [outputNames{i},"_units"], outputUnits{i});
end

(...)
```

Listing I.9: Example 2 - try-catch and console messages

```matlab
function result = mGeoGridVar(varName, mDatasetObj)
import msstate.cstm.data.grid.JGeoGridDataset
    if nargin < 2 && nargout < 1
        disp("check input and output arguments!");
        help mGeoGridVar;
        return;
    end
    if nargout > 0
        result = [];
    end

    (...)

    try
        switch nargin
            case 2
                ncID = getJDataset(mDatasetObj);
                theStruct.myNCid = ncID;
                switch class(varName)
                    case "char"
                        theStruct.varName = char(varName);
                        GeoGridData = JGeoGridDataset(ncID.getGridDataset(),
                            varName);
                        theStruct.myGridID = GeoGridData;
                        theStruct.myVarID = ncID.getJNetcdfDataset().
                            getVariable(varName);
                        myGeoGrid = GeoGridData.getGeoGrid();    % get geogrid
                    otherwise, error("MATLAB:mGeoGridVar",...
                                "varName: Input type char/string");
                end
            otherwise, error("MATLAB:mGeoGridVar:Nargin",...
                                "Incorrect number of arguments");
        end
        if (isa(myGeoGrid, "ucar.nc2.dt.grid.GeoGrid")) %check for GeoGrid
            Object
            theStruct.myShape = double(transpose(myGeoGrid.getShape()));
            result=class(theStruct,"mGeoGridVar");  %create mGeoGridVar object
        else
            result=[];
            error("MATLAB:mGeoGridVar",...
                    "Non-gridded variable. Unable to create mGeoGridVar Object
                        ");
        end
    catch %gets the last error generated
        err = lasterror();
        disp(err.message);
    end
end
```

Listing I.10: Example 1 - try-catch and file i/o

```
function [X,fk,t]=cqt_fw(audio_name)

(...)

try
    load cqt_fw_params.mat
catch ME1
    fprintf("Arquivo cqt_fw_param.mat nao encontrado, Rode a funcao
        cqt_fw_precompute\n");
    exit;
end

(...)

try
    load cqt_fw_filter.mat
catch ME2
    fprintf("Arquivo cqt_fw_filter.mat nao encontrado. Gerando os coeficientes
         dos filtros.\n");
    c1=0.3;
    c2=2.0/3;
    Q=1/(c1*(power(2,1/precisao)-power(2,-1/precisao)));
    for i=1:length(k)
        if i~=1 && i~=length(k)
            Wp=[fk0a1(i)-c1*(fk0a1(i)-fk0a1(i-1)) fk0a1(i)+c1*(fk0a1(i+1)-
                fk0a1(i))];
            Ws=[fk0a1(i)-c2*(fk0a1(i)-fk0a1(i-1)) fk0a1(i)+c2*(fk0a1(i+1)-
                fk0a1(i))];
        end
        if i==1
            Wp=[fk0a1(i)-c1*(fk0a1(i)-fk0a1_prev) fk0a1(i)+c1*(fk0a1(i+1)-
                fk0a1(i))];
            Ws=[fk0a1(i)-c2*(fk0a1(i)-fk0a1_prev) fk0a1(i)+c2*(fk0a1(i+1)-
                fk0a1(i))];
        end
        if i==length(k)
            Wp=[fk0a1(i)-c1*(fk0a1(i)-fk0a1(i-1)) fk0a1(i)+c1*(fk0a1_next-
                fk0a1(i))];
            Ws=[fk0a1(i)-c2*(fk0a1(i)-fk0a1(i-1)) fk0a1(i)+c2*(fk0a1_next-
                fk0a1(i))];
        end
        Rp=0.1;
        Rs=20;
        [N,Wn] = cheb1ord(Wp, Ws, Rp, Rs);
        [b(i,:),a(i,:)] = cheby1(N,Rp,Wn);
    end
end

(...)
```

```

if aplicar_ganho
    try
    load cqt_fw_gain;
    for i=1:length(k)
        X0(i,:)=X0(i,:)*1.0/power(gain(i),1);
    end
    catch ME3
        fprintf("nao foi possivel aplicar o ganho, arquivo cqt_fw_gain.mat nao
             encontrado\n");
    end
end

(...)

end
```

Listing I.11: Example 2 - try-catch and file i/o

```
function [dem,dax,ziro]=sergeicol
if ~exist("sergeim")
  try
    load(fullfile(getenv("IFILES"),"COLORMAPS","sergeim"))
  catch
    load("sergeim")
  end
end
reso=128;
demreso=nan(reso+1,3);
for index=1:3
  [demreso(:,index),req]=...
      discinter(sergeil,sergeim(:,index),...
                 unique([0 linspace(min(sergeil),max(sergeil),reso)]));
end
dem=demreso;
dax=minmax(sergeil);
ziro=find(req==0);
```

# Annex II

# SQL Schema and Queries

Listing II.1: Partial schema of the intelligent repository

```sql
CREATE TABLE concerns (
  id_concern INTEGER NOT NULL,
  name_concern TEXT  NOT NULL,
  desc_concern TEXT NOT NULL,
  PRIMARY KEY (id_concern)
) WITHOUT ROWID;

CREATE TABLE block_types (
  id_block_type INTEGER NOT NULL,
  name_block_type TEXT  NOT NULL,
  desc_block_type TEXT NOT NULL,
  PRIMARY KEY (id_block_type)
) WITHOUT ROWID;

CREATE TABLE toolboxes (
  id_toolbox INTEGER NOT NULL,
  name_toolbox TEXT NOT NULL,
  PRIMARY KEY (id_toolbox)
) WITHOUT ROWID;

CREATE TABLE tokens (
  id_token INTEGER NOT NULL,
  id_concern INTEGER NOT NULL,
  name_token TEXT UNIQUE NOT NULL,
  desc_token TEXT NOT NULL,
  PRIMARY KEY (id_token),
  FOREIGN KEY (id_concern) REFERENCES concerns
) WITHOUT ROWID;

```

```sql
CREATE TABLE mfiles (
  id_mfile INT(10) NOT NULL,
  id_toolbox INTEGER NOT NULL,
  name TEXT NOT NULL,
  PRIMARY KEY (id_mfile),
  FOREIGN KEY (id_toolbox) REFERENCES toolboxes
) WITHOUT ROWID;

CREATE TABLE blocks_mfiles (
  id_block INTEGER NOT NULL,
  id_type INTEGER NOT NULL,
  id_parent_block INTEGER NOT NULL,
  PRIMARY KEY (id_block),
  FOREIGN KEY (id_type) REFERENCES block_types (id_block_type),
  FOREIGN KEY (id_parent_block) REFERENCES blocks_mfiles (id_block)
) WITHOUT ROWID;

CREATE TABLE lines_mfiles (
  id_line INTEGER NOT NULL,
  line INTEGER NOT NULL,
  code TEXT NOT NULL,
  id_mfile INTEGER NOT NULL,
  PRIMARY KEY (id_line),
  FOREIGN KEY (id_mfile) REFERENCES mfiles (id_mfile)
) WITHOUT ROWID;

CREATE TABLE lines_comments (
  id_comment INTEGER NOT NULL,
  id_mfile INTEGER NOT NULL,
  line INTEGER NOT NULL,
  code TEXT NOT NULL,
  PRIMARY KEY (id_comment),
  FOREIGN KEY (id_mfile) REFERENCES mfiles (id_mfile)
) WITHOUT ROWID;

CREATE TABLE lines_tokens (
  id_line INTEGER NOT NULL,
  id_block INTEGER NOT NULL,
  id_token INTEGER NOT NULL,
  ci INTEGER NOT NULL,
  cf INTEGER NOT NULL,
  PRIMARY KEY (id_line, id_token, ci),
  FOREIGN KEY (id_token) REFERENCES tokens,
  FOREIGN KEY (id_line) REFERENCES lines_mfiles,
  FOREIGN KEY (id_block) REFERENCES blocks_mfiles
) WITHOUT ROWID;
```

Listing II.2: Repository content - files, tokens, lines and blocks

```sql
select count(*) from mfiles;

select count(*) from lines_tokens;

select count(*) from tokens;

select count(*) from lines_tokens inner join tokens using(id_token) where
    id_concern between 1 and 35;

select count(*) from lines_mfiles;

select count(*) from blocks_mfiles;
```

Listing II.3: Repository content - amount of blocks of each type

```sql
select id_type, count(id_block) from blocks_mfiles inner join block_types on
    id_type = id_block_type group by id_type;
```

Listing II.4: Repository content - block and token co-occurrence

```sql
select id_token from tokens where name like 'nargin'
-- nargin = 1125
-- iscell = 1175
-- isa = 1181
-- isstruct = 1895
-- ischar = 1439
-- disp = 558
-- error = 847

-- Co-occurrences --
select count(id_block) from blocks_mfiles inner join lines_tokens using(
    id_block) where id_type = 3 and id_token = 1125
-- 3|if
-- 4|while
-- 5|for
-- 6|switch-case
-- 7|parfor
-- 8|spmd
-- 9|try-catch

-- Token occurrences --
select count(*) from lines_tokens where id_token = 1125
```

Listing II.5: Number of files with for if and DTV

```sql
select count(distinct id_mfile)
from blocks_mfiles as p
inner join (
    select distinct id_block, id_parent_block
    from blocks_mfiles inner join lines_tokens using(id_block) inner join
       tokens using(id_token)
    where id_type = 3 and id_concern between 3 and 4
) as c
on p.id_block = c.id_parent_block
where p.id_type = 5
```

Listing II.6: Number of files with while and Sys-CF

```sql
with parents as
(
    select id_block, id_type, id_parent_block
    from blocks_mfiles
    where id_type = 4
    union all
    select b.id_block, b.id_type, b.id_parent_block
    from blocks_mfiles b inner join parents p
    on b.id_parent_block = p.id_block
)
select count(distinct id_mfile) FROM blocks_mfiles inner join parents
    using(id_block) inner join lines_tokens using(id_block) inner join
    tokens using(id_token) where id_concern = 23
```

Listing II.7: Number of files with switch and VFArg

```sql
select distinct id_mfile
from blocks_mfiles inner join lines_tokens using(id_block) inner join
    tokens using(id_token)
where id_concern = 1 and id_type = 6
```

Listing II.8: Number of files with try-catch and CM

```sql
select distinct id_mfile
from blocks_mfiles inner join lines_tokens using(id_block) inner join
    tokens using(id_token)
where id_type = 9 and id_concern = 6
```

Listing II.9: Number of files with try-catch and FIO

```sql
select distinct id_mfile
from blocks_mfiles inner join lines_tokens using(id_block) inner join
    tokens using(id_token)
where id_type = 9 and id_concern = 21
```

Listing II.10: Number of files with if+nargin blocks

```sql
select count(distinct id_mfile)
from blocks_mfiles inner join lines_tokens using(id_block)
where id_token = 1125 and id_type = 3
```

Listing II.11: Number of directly associated concern tokens to if+nargin blocks

```sql
select count(*)
from (
    select distinct id_block from blocks_mfiles inner join lines_tokens
        using(id_block)
    where id_token = 1125 and id_type = 3
)
inner join lines_tokens using(id_block) inner join tokens using(id_token)
where id_concern between 1 and 35;
```

Listing II.12: Number of hierarchically associated concern tokens to if+nargin blocks

```sql
with parents as
(
    select id_block, id_type, id_parent_block
    from (
        select distinct id_block, id_type, id_parent_block
        from blocks_mfiles inner join lines_tokens using(id_block)
        where id_token = 1125 and id_type = 3
    )
    union all
    select b.id_block, b.id_type, b.id_parent_block
    from blocks_mfiles b inner join parents p on b.id_parent_block = p.
        id_block
)
select id_concern, count(id_token)
from parents inner join lines_tokens using(id_block) inner join tokens
    using(id_token)
group by id_concern
```

Listing II.13: Number of files with switch+nargin blocks

```
select count(distinct id_mfile) from blocks_mfiles inner join lines_tokens
     using(id_block)
where id_token = 1125 and id_type = 6
```

Listing II.14: Number of directly associated concern tokens to switch+nargin blocks

```
select count(*)
from (
    select distinct id_block from blocks_mfiles inner join lines_tokens
        using(id_block)
    where id_token = 1125 and id_type = 6
)
inner join lines_tokens using(id_block) inner join tokens using(id_token)
where id_concern between 1 and 35;
```

Listing II.15: Number of hierarchically associated concern tokens to switch+nargin blocks

```
with parents as
(
    select id_block, id_type, id_parent_block
    from (
        select distinct id_block, id_type, id_parent_block
        from blocks_mfiles inner join lines_tokens using(id_block)
        where id_token = 1125 and id_type = 6
    )
    union all
    select b.id_block, b.id_type, b.id_parent_block
    from blocks_mfiles b inner join parents p on b.id_parent_block = p.
        id_block
)
select id_concern, count(id_token)
from parents inner join lines_tokens using(id_block) inner join tokens
    using(id_token)
group by id_concern
```

Annex III

# Supplementary Data

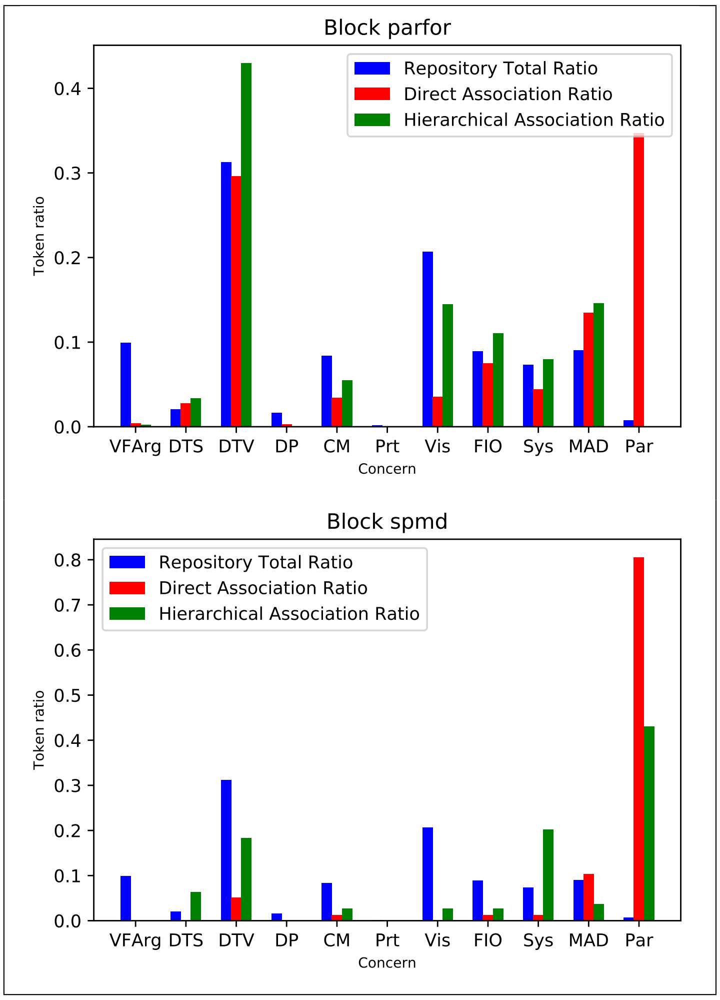


Figure III.1: Token ratio vs Concern - Parallel Statement Blocks

Table III.1: Mapping between Concerns and corresponding Tokens [8]

| | |
|---|---|
| 1. Verification of function arguments and return values | nargchk, nargin, nargout, nargoutchk, varargin, varargout |
| 2. Data type specialization | fi, fimath, int16, int32, int64, int8, quantize, quantizer, sfi, single, ufi, uint16, uint32, uint64, uint8, double |
| 3. Data type verification | cast, class, intmax, intmin, isa, isboolean, iscell, ischar, iscolumn, isempty, isfi, isfield, isfimath, isfixed, isfloat, isinf, isinfinite, isinteger, islogical, isnan, isnumeric, isobject, isquantizer, isreal, isrow, isscalar, isstr, isstruct, isvector, length, ndims, numel, range, realmax, realmin, size, typecast, wordlength |
| 4. Dynamic properties | eval, evalc, evalin, inline, feval |
| 5. Console messages | annotation, assert, disp, display, error, last, lastwarn |
| 6. Printing | orient, print, printdlg, printopt |
| 7. Visualization | aaxes, axis, box, cla, clabel, clf, close, datacursormode, datetick, errorbar, figure, figurepalette, fplot, frame2im, gca, gcbf, gcbo, gco, getframe, gplot, grid, gtext, hist, histogram, hold, im2frame, image, imfinfo, imformats, imread, imwrite, ishold, legend, line, loglog, mesh, meshgrid, movie, newplot, pan, plot, plot3, plotbrowser, plotedit, plottools, plotyy, polar, propertyeditor, rectangle, reset, rgbplot, rotate, rotate3d, scatter, semilogx, semilogy, set, showplottool, subplot, surf, texlabel, text, title, VideoReader, VideoWriter, xlabel, ylabel, zlabel, zoom |
| 8. File I/O | diary, fgetl, fgets, fileformats, fopen, fprintf,fread, fscanf, fwrite, hgload, hgsave, load, save, saveas, uisave |
| 9. System | addtodate, ans, ba, bafter, batch, break, calendar, calllib, clear, clearvars, clock, cputime, date, dbcont, dbmex, dbquit, dbstop, ebreak, echo, etime, exist, inmem, input, inputname, inputParser, isglobal, iskeyword, isvarname, libfunctionsview, libisloaded, loadlibrary, memory, mex, mexext, mfilename, mislocked, mlock, munlock, namelengthmax, nanbreak, next, now, onCleanup, pack, pause, pcode, rbreak, rehash, run, slist, spmd, start, startat, step, stop, symvar, systems, tbreak, tic, timerfind, timerfindall, toc, unloadlibrary, wait, weekday, where, who, whos, xbreak, zcbreak |
| 10. Memory allocation/ deallocation | delete, global, ones, persistent, zeros |
| 11. Parallelization | cancel, codistributed, codistributor, createParallelJob, createTask, defaultParallelConfig, demote, destroy, detupForParallelExecution, dfeval, dfevalasync, distributed, gather, gcat, gop, gplus, gpuArray, gpuDevice, gpuDeviceCount, importParallelConfig, isreplicated, jobStartup, labBarrier, labBroadcast, labindex, labProbe, matlabpool, mpiLibConf, mpiprofile, mpiSettings, numlabs, parfor, pctconfig, pctRunDeployedCleanup, pctRunOnAll, pload, pmode, poolStartup, promote, psave, redistribute, resume, sparse, submit, subsasgn, subsref, taskFinish, taskStartup |

Listing III.1: GitHub project fetcher

```python
#!/usr/bin/env python3
"""GitHub project fetcher
REQUIRES: python3
REQUIRES: Requests (pip/pip3 install requests)
REQUIRES: Unix environment (common utility binaries)

Automatically fetches GitHub repositories following certain criteria.
See -h for usage info.
See https://developer.github.com/v3/search/#search-repositories for query parameter specifications.
Example usage:
    ./getgit.py -n 200 -q "language:matlab" "size:<100" -s stars
"""
import argparse
import logging as log
import requests
import json
import subprocess
import traceback
import re
from time import sleep


API_URL = "https://api.github.com/search/repositories?q="
REPO_ARCH_SUFFIX = "/archive/master.zip"


def getResults(parms, sort, order, perPage, n, nt, wt):

    results = []

    nextReq = API_URL + "+".join(parms) + ("&sort=" + sort if sort else "") + ("&order=" + order if order else "") + ("&per_page=" + str(perPage) if perPage else "")

    while len(results) < n:

        if not nextReq:
            debug.info("No more items!")
            break

        numRetries = -1

        while True:
            resp = requests.get(nextReq)
            log.debug("Made request: " + nextReq + "(" + str(resp.status_code) + ")")


            if resp.status_code != 200:
                numRetries += 1
                log.info("Request failed with status code " + str(resp.
                    status_code) + " (probably rate limited)")

                if numRetries >= nt:
                    break

                # Wait and try again
                log.info("Retrying in " + str(wt) + "s")
                sleep(wt)

            else:
                break

        # Reached limit, return what we have
        if numRetries >= nt:
                log.error("Reached maximum retries; exiting")
                break


        # Get next page
        hds = resp.headers["Link"].split(",")
        for s in hds:
            url = re.search(r"<(.+)>; rel=\"next\"", s)

            if url:
                nextReq = url[1]
                break


        results += resp.json()["items"]


    return results[:n]

def dlRepos(items, archProc, finalProc):
    log.info("Starting retrieval")

    lenR = len(items)
    i = 0
    for item in items:
        try:
            i += 1
            log.info("Retrieving %s (%d/%d)", item["full_name"], i, lenR)

            url = item["html_url"] + REPO_ARCH_SUFFIX
            subprocess.run(["wget", url, "-O", "tmp.zip", "-q", "--show-
                progress"], check=True, stdout=subprocess.DEVNULL)

```

```python

            if not archProc("tmp.zip"):
                log.info("Exiting after processing compressed repo (processor returned False)")

                subprocess.run(["rm", "tmp.zip"], check=True, stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
                continue


            log.info("Unzipping")


            # Unzip repo, rename to friendlier version (full name, / replaced by -)
            subprocess.run(["unzip", "tmp.zip"], check=True, stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
            subprocess.run(["rm", "tmp.zip"], check=True, stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
            name = item["full_name"].replace("/", "-")
            subprocess.run(["mv", item["name"] + "-master", name], check=True, stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)


            if not finalProc(name):
                log.info("Exiting after processing uncompressed repo (processor returned False)")

                assert(name.find("..") == -1)
                assert(not re.search(r"\s", name))
                subprocess.run(["rm", "-rdf", name], check=True, stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
                continue

            log.info("Done")
        except Exception:
            log.error("", exc_info=True, stack_info=True)
            log.error("Exception during repository handling; continuing")


def main():

    """Processor for JSON results.
    main() will discard this result if it returns False.

    e.g.:
        if result["name"] == "fooRepoName": return False
        return True
    """
    def resProc(result):
        return True

    """Processor for retrieved repositories which runs before inflation.
    Args: archive filepath;
    Example usage: removing unwanted files
    dlRepos() will skip this repository if it returns False.
    """
    def archProc(archPath):
        return True

    """Processor for retrieved repositories which runs after inflation.
    Args: repository directory filepath;
    dlRepos() will skip this repository if it returns False.
    """
    def finalProc(repoPath):
        return True


    args = getArgs()

    loglevel = args.log
    nLoglevel = getattr(log, loglevel)
    log.basicConfig(level=nLoglevel, format="%(asctime)s: %(levelname)s: %(
        message)s", datefmt="%H:%M:%S %d-%m-%Y")
    log.info("Started at loglevel %s", loglevel)

    parms = args.query
    log.info("Getting results for %s", parms)

    res = None
    if args.use_results:
        log.info("Fetching from " + args.use_results)
        with open(args.use_results, "r") as f:
            res = json.load(f)
    else:
        res = getResults(parms, args.sort, args.order, args.per_page, args.n,
            args.nt, args.wt)

        if args.fetch_results:
            log.info("Dumping results to " + args.fetch_results)
            with open(args.fetch_results, "w") as f:
                json.dump(res, f, indent=2)

            return


    log.info("Got " + str(len(res)) + " results")

    res[:] = [r for r in res if resProc(r)]
    log.info(str(len(res)) + " results after processing")
```

```python
    log.debug("Results:\n%s", res)

    dlRepos(res, archProc, finalProc)
    log.info("Exiting")


def integer_positive(a):
    v = int(a)
    if v <= 0:
        raise argparse.ArgumentTypeError(str(a) + " must be a positive integer")
    return v

def getArgs():
    parser = argparse.ArgumentParser()

    parser.add_argument("--log", type=str, help="logging level", choices=["DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"], default="INFO")
    parser.add_argument("-q", "--query", type=str, help="query parameters", nargs="+", required=True)
    parser.add_argument("-s", "--sort", type=str, help="sort function", default="")
    parser.add_argument("-o", "--order", type=str, help="order function", default="")
    parser.add_argument("-p", "--per-page", type=integer_positive, help="results per page", default=100)
    parser.add_argument("-n", type=integer_positive, help="number of repositories to retrieve", default=1000)
    parser.add_argument("-nt", type=integer_positive, help="number of request retries", default=1)
    parser.add_argument("-wt", type=integer_positive, help="time in seconds between retries", default=60)

    results = parser.add_mutually_exclusive_group()
    results.add_argument("--fetch-results", type=str, help="fetch results only and dump to file")
    results.add_argument("--use-results", type=str, help="use results from file")

    return parser.parse_args()


if __name__ == "__main__":
    main()
```

Listing III.2: Non-m-file remover

```
#!/usr/bin/env python3
"""
Module to remove files that do not end with the *.m file extension. Can be
    used as standalone script or be imported into existing script.
"""
import os

def main():
    for root, dirs, files in os.walk(".", topdown=False):
        for name in files:
            if not name.endswith(".m"):
                os.remove(os.path.join(root, name))
      elif name.startswith("._"):
        os.remove(os.path.join(root, name))

if __name__ == "__main__":
    main()
```

Listing III.3: Empty folder remover

```python
#! /usr/bin/env python
"""
Module to remove empty folders recursively. Can be used as standalone script
    or be imported into existing script.
"""
import os, sys

def removeEmptyFolders(path, removeRoot=True):
  'Function to remove empty folders'
  if not os.path.isdir(path):
    return

  # remove empty subfolders
  files = os.listdir(path)
  if len(files):
    for f in files:
      fullpath = os.path.join(path, f)
      if os.path.isdir(fullpath):
        removeEmptyFolders(fullpath)

  # if folder empty, delete it
  files = os.listdir(path)
  if len(files) == 0 and removeRoot:
    print "Removing empty folder:", path
    os.rmdir(path)

def usageString():
  'Return usage string to be output in error cases'
  return 'Usage: %s directory [removeRoot]' % sys.argv[0]

if __name__ == "__main__":
  removeRoot = True

  if len(sys.argv) < 1:
    print "Not enough arguments"
    sys.exit(usageString())

  if not os.path.isdir(sys.argv[1]):
    print "No such directory %s" % sys.argv[1]
    sys.exit(usageString())

  if len(sys.argv) == 2 and sys.argv[2] != "False":
    print "removeRoot must be 'False' or not set"
    sys.exit(usageString())
  else:
    removeRoot = False

  removeEmptyFolders(sys.argv[1], removeRoot)
```