



INSTITUTO
UNIVERSITÁRIO
DE LISBOA

PescaJ: A Projectional Editor for Java with Overlapping Abstract Syntax Tree Views for Code and Documentation

José Miguel Faustino Lopes

Master in Computer Science and Engineering

Supervisor:

Prof. Dr. André Leal Santos, Assistant Professor
Iscte - Instituto Universitário de Lisboa

October, 2023



TECNOLOGIAS
E ARQUITETURA

Department of Information Science and Technology

PescaJ: A Projectional Editor for Java with Overlapping Abstract
Syntax Tree Views for Code and Documentation

José Miguel Faustino Lopes

Master in Computer Science and Engineering

Supervisor:

Prof. Dr. André Leal Santos, Assistant Professor
Iscte - Instituto Universitário de Lisboa

October, 2023

Resumo

Convencionalmente, código fonte, incluindo documentação, é simultaneamente uma representação de armazenamento e edição, através de ficheiros e editores que os manipulam como texto. Ao longo dos anos, os IDEs tornaram-se mais sofisticados, fornecendo funcionalidades que adicionam informação útil ao texto base (e.g., *inlay type hints*), por outro lado, procuram a diminuição de informação irrelevante, reduzindo *clutter* (e.g., *code folding*). Estas tendências são um sinal que os utilizadores procuram maneiras mais convenientes de editar código em relação à manipulação direta de texto.

Elaboramos um estudo focado em motivar o design de um editor com capacidades de navegação e edição de código e documentação superiores à manipulação de ficheiros. Este estudo aborda estruturas de código, nomeadamente interações entre métodos, como número de métodos chamados. Também foi medida a quantidade de documentação, quantificando o impacto de *clutter* que a documentação causa na escrita de código.

Apresentamos o PescaJ, um protótipo de editor projecional para projetos Java que diverge da edição de código orientada ao ficheiro fonte, possibilitando a criação de vistas que agregam métodos que pertencem a classes distintas, estes podem estar presentes e ser editados em múltiplas vistas. O editor coloca automaticamente os fragmentos de código na interface, justapondo os mesmos em função das relações que estes apresentam no código. Para além disto, a ferramenta oferece vistas de documentação, agregando comentários Javadoc normalmente espalhados pelos ficheiros. Estas reduzem *clutter* e aliviam o utilizador da sintaxe, também separando as tarefas de escrita de código e documentação, podendo reduzir o impacto cognitivo no utilizador.

Palavras Chave: Editor Projecional, Separação de Conceitos, Documentação, Java

Abstract

Conventionally, source code is simultaneously a storage and editing representation, through files and editors to manipulate them as text. Over the years, IDEs have become increasingly sophisticated, providing features to augment the visible text content with helpful information (e.g., overlay documentation popups, inlay type hints), or on the opposite, to decrease it to reduce clutter (e.g., code folding on documentation, methods, etc.). This is a sign that developers seek more convenient code editing forms than the direct manipulation of text files.

We conducted an empirical study that seeks to motivate the design of an editor which enhances code and documentation navigation and editing, beyond source file manipulation. The study addresses code structure, namely method interaction, such as number of methods called and depth of call graph. We also measured the amount of documentation present in source files, quantifying the amount of clutter documentation causes while coding.

We present PescaJ, a prototype projectional editor for Java projects that breaks away from file-oriented source code editing, providing the possibility of forming views that aggregate methods that belong to different classes, where single methods may be simultaneously present and edited in multiple views. It features automatic code fragment placement, juxtaposing them according to their interactions in the code. Furthermore, PescaJ provides documentation editors, also aggregating scattered Javadoc comments. The specialized documentation views reduce clutter and relieve the user of syntax, while separating the task of writing code and its documentation, separating the two technical *concerns*, that may lower cognitive strain on the user.

Keywords: Projectional Editor, Separation of Concerns, Documentation, Java

Contents

CHAPTER 1. Introduction.....	7
1.1. Background and Motivation	7
1.2. Goals.....	8
1.3. Research Questions.....	9
1.4. Contributions	9
1.5. Research Method.....	10
CHAPTER 2. State of the Art	13
2.1. Architecture	13
2.1.1. Abstract Syntax tree	13
2.1.2. Source Based Editing	14
2.1.3. Structured Code Editors	15
2.1.4. Model-View-Controller (MVC).....	15
2.1.5. Projectional Editing.....	16
2.2. Human-Computer Interaction	17
2.2.1. Fitt’s Law	17
2.2.2. Split Attention Effect and Spatial Contiguity	18
2.3. Related Tools.....	20
2.3.1. CodeBubbles.....	20
2.3.2. PatchWorks.....	21
2.3.3. Barista.....	23
2.3.4. Javardise	24
2.4 Behavior of Text-based Code Editors.....	25
CHAPTER 3. Empirical Study of Method Dependencies and Documentation.....	29
3.1. Metrics	29
3.2. Qualitas Corpus.....	31
3.3. Package <i>java.util</i>	32
3.4. Results.....	33
3.4.1. Method Calls.....	33
3.4.2. Call Depth	36
3.4.3. Lines of Documentation	39
CHAPTER 4. PescaJ.....	41
4.1. MVC architecture with overlapping views.....	41
4.2. Design.....	42
4.2.2. Documentation views.....	45
4.3. Implementation	48
4.4. Informal Evaluation.....	48

4.4.1. Evaluation of code navigation	48
4.4.2. Evaluation of documentation navigation	49
CHAPTER 5. Conclusions and Future Work	51
5.1. Future Work	52
5.1.1. Save State	52
5.1.2. In-editor run-time	52
5.1.3. New Views	53
5.1.4. Enabling user customization.....	53
5.1.5. Code Elision	53
5.1.6. User Testing	54
References.....	55

CHAPTER 1.

Introduction

Developers find themselves interacting with previously written code a large chunk of the working day, with initial development of applications taking up only a small fraction of development time [18]. It has been measured that developers spend up to 70% of their time understanding existing code, 17% interacting with the user interface of the IDE and as little as 5% editing code [22].

Conventional text-based editors use the source code simultaneously as storage and editing. Over the years these types of IDEs have seen an increase in features that augment text visibility or navigation (e.g., overlay documentation popups, inlay type hints), or on the opposite, to decrease it to reduce clutter (e.g., code folds on imports, documentation, methods, etc). This trend is a sign that developers seek ways to more conveniently navigate and visualize code, rather than the direct manipulation of text files.

1.1. Background and Motivation

Separation of concerns is a classical design principle in software engineering that advocates decomposing systems into artifacts according to distinct concerns. The general principle is somewhat vague and what is exactly a concern is a subject of multiple interpretations. Concerns are often discussed in terms of domain logic and its decomposition into implementation artifacts. In this context, the notion of scattering refers to domain features whose implementation is spread over multiple code artifacts, whereas tangling results from mixing parts of the implementations of different features in the same code artifact.

Yip & Lam [31] state that for tasks that require visual comparisons of elements that can't be stored in visual memory, multiple views should be employed, rather than using tabs or zoom. Arguably, software development is one of these tasks, developers have to constantly look at multiple complex classes or methods to understand previously written code. Because of scattering of implementation artifacts in text files, there is no simple way to have every needed visual information on the screen at the same time, with text-based code editors, since we are constrained by the structure of the text file. This forces contextually relevant information to be placed in separate windows [15]. Because of this limitation, developers have to not only scroll up and down the file, but also switch between source files.

Conventionally, code artifacts hold both implementation and documentation in source code files. This is convenient for associating pieces of documentation with the respective implementation elements. On the other hand, documentation often contains a large number of lines, greatly increasing clutter and length of the source code, causing the need for longer scrolls and *scattering* code fragments further. These effects are burdensome for developers, which clearly seek solutions, seeing as most modern tools have the capability of collapsing documentation. We consider that code and its documentation are *tangled* in the source files, and as such, that these two fragments are separate *concerns*.

A factor that may have an impact on the time and efficiency of understanding code, is the dichotomy between the inherent structure of code dependencies and their representation in the source files. When code is represented in text files, the order of the code elements does not necessarily reflect their dependencies. Furthermore, closely related elements of code can be kept in completely separated source files, making it an inefficient way to visualize code [16].

1.2. Goals

Motivated by the huge part that understanding plays in the activity of a software developer, and the growing search for features that reduce the burden of this task, the goal of this dissertation work was to develop a code editor that helps developers save time on navigation and understanding of code and its documentation. This tool seeks to display the graph-like structure of the code to help its understanding, while separating the concern of documentation, which is aggregated in specialized views. We also took into consideration the cognitive and ergonomic strain that having closely related information be separated (by tabs or scroll-bars) takes on the user [6, 7, 27, 31]. To avoid this, the editor displays juxtaposed information, seeking to reduce navigation time and lighten the cognitive load on the user.

To motivate the development of the aforementioned editor, a small study was conducted on a large curated collection of open-source Java systems, *Qualitas Corpus* [28] and the open-source Java library *java.util*. The study seeks to identify common structure patterns, considering intra- and inter-class dependencies.

We developed a Projectional Editor for Scattered Code (and documentation) Aggregation for Java, PescaJ.

1.3. Research Questions

- Q1. How are the intra- and inter-class dependencies characterized in existing projects?
- Q2. How to design an interface that organizes classes and methods according to their dependencies?

1.4. Contributions

Part of the content of this dissertation is based on the following paper [19], which describes the main ideas of PescaJ:

PescaJ: A Projectional Editor for Java Featuring Scattered Code Aggregation

José F. Lopes, André L. Santos

Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT),

SPLASH'23 Workshops, October 23, 2023, Cascais, Portugal.

The first step to the development of PescaJ was to investigate code structure to identify the most common patterns. This provided an aid for making decisions for the behavior of the tool. The study was conducted as an empirical study on the complexity of the dependency tree of a large curated collection of open-source Java systems, *Qualitas Corpus* [28], as well as the *java.util* package from Java standard libraries, identifying common structure patterns.

A proof-of-concept tool was developed based on the results of the developed study and following some human-machine interaction principles. The developed tool, PescaJ, is a projectional editor that seeks to automatically juxtapose closely related fragments of code, pursuing the following:

- **Reduce navigation operations:** Because the tool displays relevant information concurrently on screen, the user should have to perform less scrolling and tabbing actions to reach needed fragments of information.
- **Fewer navigation errors:** Not only does less navigation (1) bring fewer overall errors, the proximity of the fragments can also reduce wrong clicks caused by human error.
- **Improved cognitive load:** Because of the Split Attention Effect, having closely related information be clustered together, could have a beneficial effect on the user's working memory load.
- **Improved/Faster code understanding:** PescaJ displays code logic more obviously when compared to mainstream text-based code editors, improving its understanding. The effect of (3) should also have an impact on this front.

- Enhanced Documentation:** PescaJ offers the capability of separating code and its documentation, not only reducing the clutter caused by documentation while coding, but also enhancing the task of writing documentation. The aggregation and juxtaposition of documentation, makes the task of creating consistent documentation with respect to terminology and style easier.

PescaJ is a Java editor that was developed using Kotlin, closely following the model-view-controller architectural pattern. JavaParser [3] was used for the parsing of Java code into an in-memory workable model, and Javardise [26] was used as the default view for code editing. PescaJ can be found on Github¹.

1.5. Research Method

The research method used in this work was the Design Science Research Process (DSRP) [24], it is a methodology focused on the development of a solution for a problem and is widely used in engineering and computer science. This research method employs a loop of research, development and evaluation, resulting in an iterative process that provides a better artifact with each pass. Figure 1.1 illustrates the distinct steps of DSRP, highlighting possible entry points of the methodology.

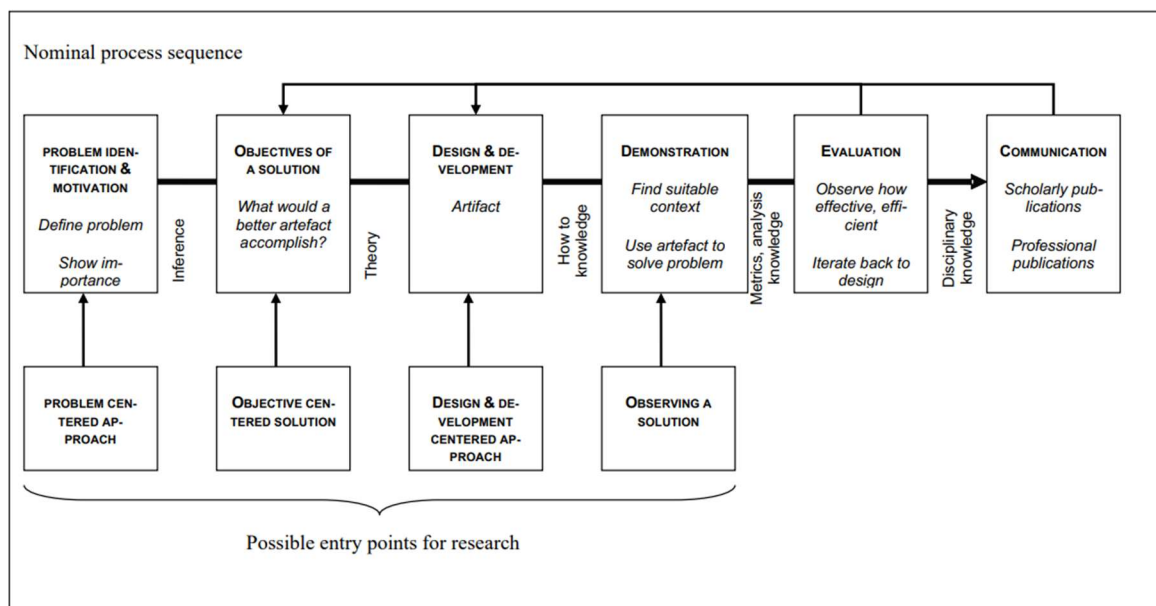


Figure 1.1: Design Science Research Process [24].

¹ <https://github.com/Jose-f-Lopes/PescaJ>

The development of this work and PescaJ can be overlapped with the DSRP steps quite clearly, first, the objectives mentioned previously characterize the desired behavior of the proposed solution, this represents the entry point for the project with the “Objective centered solution”. The second step was to research relevant topics and tools with similar objectives to those of PescaJ, together with the study realized on code based and the development of the tool itself they make up the “Design & Development” of the artifact. Using some classes from the *java.util* package from Java, namely *ArrayList*, PescaJ’s capabilities were demonstrated and compared to conventional workflows of source-based code editors. This makes up the “Demonstration” and “Evaluation” stages. Ideally, real user testing would be conducted in this phase, however it ended up being out of scope of this dissertation work. Finally, the “Communication” step consists of, not only the writing of this present thesis, but also the development and publication of a paper [19]. In Section 5.2, we dive into possible future features, enhancements and other ideas for PescaJ, these represent possible work for subsequent iterations on the DSRP methodology.

State of the Art

In this chapter we will explore previously developed works that have similar goals or implementation to the objective artifact, we will also describe some relevant theoretical themes to the project, such as projectional editing. Because of the Human-computer interaction nature of our project, we will also present an overview of some topics in this matter.

2.1. Architecture

In the architecture topic, we will analyze some theoretical concepts like the Abstract Syntax Tree, as well as explore the different types of editors. Finally, we will approach some design patterns and tools that will be used in the implementation of the project.

2.1.1. Abstract Syntax tree

An Abstract Syntax Tree (AST) is a tree representation of syntactic structured text that follows an abstract grammar. It is the result of a lexical and syntactic analysis of a source text, as an example, for code an AST is the result of parsing the source code in the compiler.

In the scope of this project, ASTs will always be referred to in the context of code. Figure 2.1 is a simplified representation of an example of an AST, showing the tree structure.

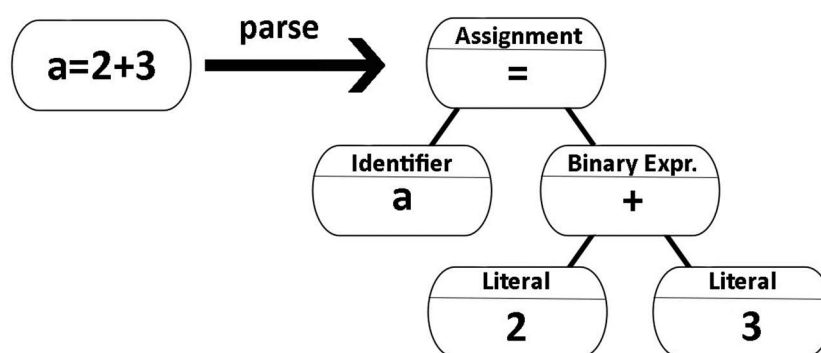


Figure 2.1: An assignment and its representation of its AST.

Because ASTs are the result of compilation, they exhibit the logic of the execution of the program much more clearly than source files. Unfortunately, the ASTs are very detailed, with things like operators taking up a whole node of the tree, which makes them relatively hard to read.

As mentioned before, ASTs are a crucial step in the compilation of executables, but their usefulness goes beyond that. For real time compilation error detection in modern IDEs, the compiler needs to build the AST as the user inputs code, since ASTs are the structure that allow the verification of the syntactic structure of the code. This is why most contemporary IDEs can only detect compilation errors moments after the user finishes inputting a code fragment.

Another very important application of ASTs is refactoring. The refactoring tool traverses the nodes of the AST and makes the necessary changes in the relevant sections. This process would not be possible with the source files exclusively, since these lack contextual information without parsing, for example, it is possible to have two variables with the same name in different contexts, such as different functions. Another use case for ASTs is the analysis of code structure and quality. Static analysis tools usually parse the source code into the AST to understand the code structure, detecting performance issues, unused code, etc.

JavaParser [3] is a tool that converts Java source code into an AST model, it allows navigation through the model, with structures like visitors, as well as editing. This tool will be a huge part of the developed project, since the model for the MVC architecture will be, in great part, created by JavaParser.

2.1.2. Source Based Editing

Source based Editing, is when a system's storage representation and editable fragments are the same, the source files. For executing the system, the source files are compiled into an abstract representation, such as the abstract syntax tree, and then converted into an executable. Figure 2.2 shows the structure of source-based editors, as well as the conversions between representations.

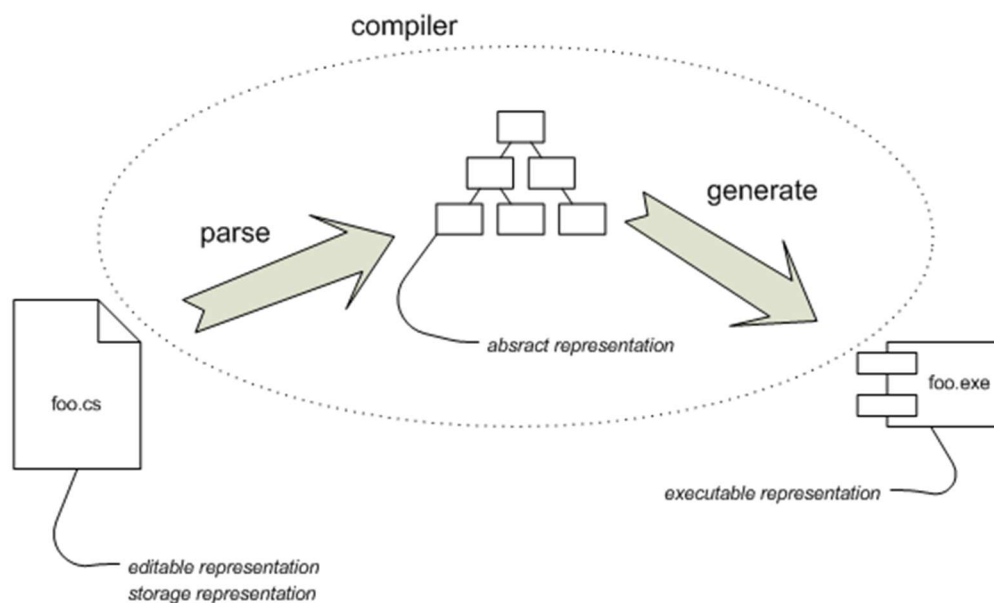


Figure 2.2: Structure of Source based editor [8].

This paradigm of programming is very widespread, making up most of the tool ecosystem. Because of how mainstream these are, there is a general consensus on the format of the source files, with most systems using text files for storage and editing. A specific case of these types of systems are text-based code editors, such as Eclipse.

2.1.3. Structured Code Editors

In contrast to editing though unstructured text, structured code editors enforce the correct syntactic structure of code though the whole programming process, this means code produced in a Structured editor will always parse. While these types of editors lack the flexibility of unstructured text, it has been shown that the majority of modifications that programmers make to code can be fully achieved by structured editing [14].

2.1.4. Model-View-Controller (MVC)

Model-View-Controller (MVC) [17] is a software design pattern, usually employed in user interfaces. This design pattern splits the program into three separate modules: model, view and controller. Figure 2.3 shows an overall diagram of the interactions between the modules defined in the design pattern, as well as the user.

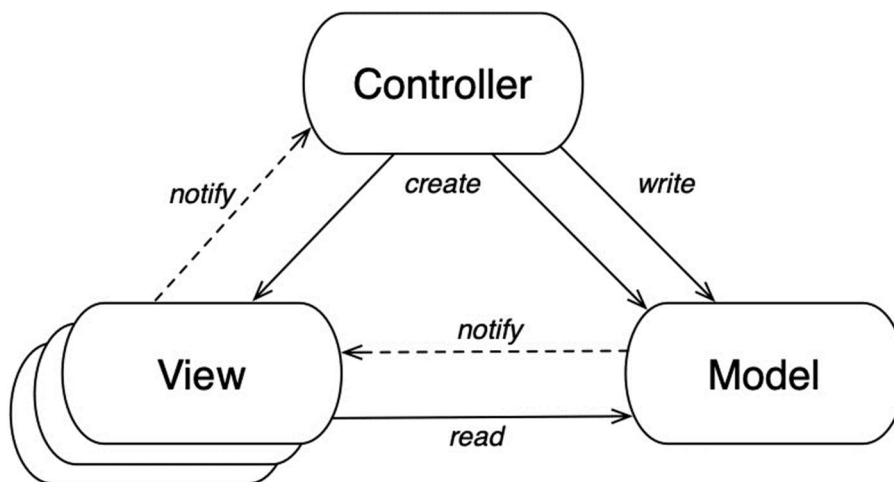


Figure 2.3: Model-View-Controller representation.

- **The model** is the main component of the application, being responsible for all the program's logic and data structure. The model is completely independent from the user interface (the views), this means that it does not need to worry about the logic of the interface, needing only to provide an application programming interface (API) for the views to collect relevant information.

- **The views** are responsible for displaying the information from the model to the user. Because the model is independent from the views, the latter can be freely programmed, regardless of the internal logic of the model, being able to display information in any format (e.g. text, tables, charts, etc.). Because views can be so diverse, they need to implement an API for the editing actions, so the controller can properly receive the information from the user inputs.
- **The controller** is responsible for affecting the model with the changes the user applies to the interface (the views). The controller is the only component that can apply changes to the model, this causes all of the editing to be centralized in the code, meaning only the controller needs to know how to properly apply changes to the model, abstracting the views.

The properties of MVC allow multiple views (of any format) to display information of the same fragment of the model, they also allow editing from these separate views without any input conflict.

2.1.5. Projectional Editing

An alternative paradigm to Source Based Editing is Projectional Editing. In this paradigm, the main definition of the system is the abstract representation, that gets directly transformed into an executable for running, and also transformed into a storage representation for permanent storage as a file. Generally speaking, projectional editing is an implementation of the MVC pattern for editing code, where the model is the AST. The following Figure 2.4 shows a general scheme of the architecture of a projectional editor.

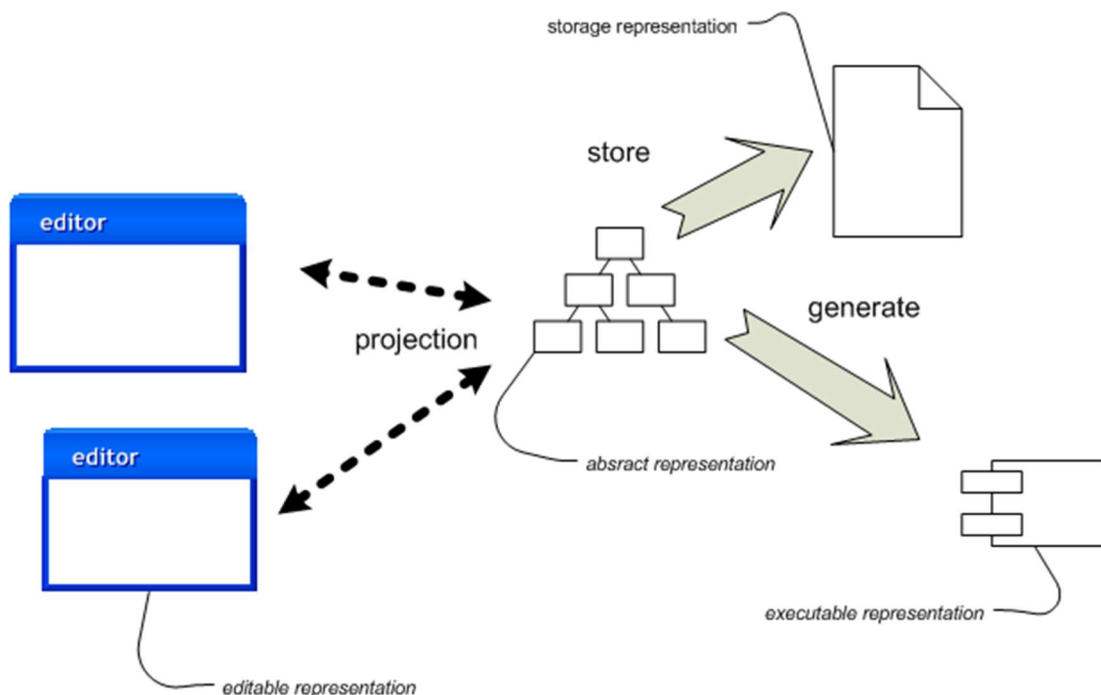


Figure 2.4: Projectional editing representation [8].

The editing of the system is done through projections, as the name suggests, they are a way to visualize and edit the core abstract representation directly. These projections do not need to follow the syntax of code, can be partial and do not have a specified form, meaning they can take the shape of text, tables, graphs, etc. [25].

Because of how free form these projections are, they can be built and adjusted to show more relevant information, and leave out unnecessary detail, like text formatting. It is even possible for users to switch between different types of projections in real time.

Seeing as the definition of the system (abstract representation) and editable fragments (projections) are separate, one given fragment of code can be edited and visualized by multiple distinct projections, at the same time. This is one of the advantages of the projection paradigm when compared to Source based Programming, seeing as the latter unifies editing and storage, making editing from multiple sources much harder.

The properties of projectional editing also have down-sides. Because the core information of the system is in the abstract representation, which is kept in quick access memory, there is not a concrete way to store the information permanently, and each implementation of this paradigm can take a slightly different approach. This makes it much harder to have a unified system that enables remote and parallel access, like Git for Source based systems.

2.2. Human-Computer Interaction

In this section, we will explore some human-computer interaction themes, namely Fitt's law, split attention effect and spatial contiguity.

2.2.1. Fitt's Law

Fitt's law is an ergonomics and human-computer interaction model that predicts the behavior of human pointing (finger, mouse, eyesight, etc.) as a function of the distance and size of the target. This model was originally studied on a 1-dimensional scenario [6, 7], where testers would have to point to targets with a stylus. However, it has been shown that this law can still be useful in 2 dimensional scenarios [20], with some adjustments.

In Figure 2.5, there is a representation of the user's pointer, as well as a target widget, the pointer and the widget are A distance apart, and the widget has width of W and height H . The model states that the higher the ratio between the distance of the target A , and its size W and H , the higher the difficulty of pointing to the target.

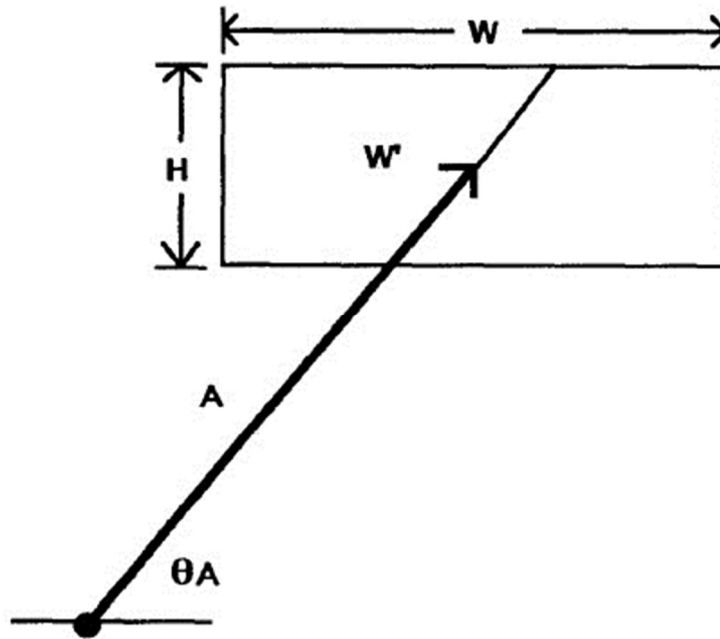


Figure 2.5: Scheme of pointer moving to target [20].

While the precise calculations provided by this model will not be used in this project, the general idea provided by this law is still very useful. As the distance between the cursor and the target increases, so does the likeliness of a pointing mistake. The same goes for the size of the target, the smaller it is, the more pointing mistakes it will cause. While notion is the basis of some commonly used user interface widgets (e.g.: drop down menus, radial menus, etc.), it is also relevant for the arrangement of views in this project's tool. According to this model, fragments of code that are often interacted with consecutively, should be arranged closer together, mitigating pointing amplitude, which lowers mistakes made in navigation.

2.2.2. Split Attention Effect and Spatial Contiguity

The split attention effect and spatial contiguity principle are closely related. Both notions state that humans learn better when relevant pieces of information are presented closer together, for example, an image and a relevant explanation subtitle, work best when they are closer in a page, rather than far apart.

In Figure 2.6 there are two alternative representations of the water cycle, on the left-hand side one, the diagram and text explanation are separated, going against the spatial contiguity principle, causing split attention effect, lowering learning efficiency. The right-hand side representation, the diagram and text are completely integrated, following the spatial contiguity principle, lowering the split attention effect and bettering understanding.

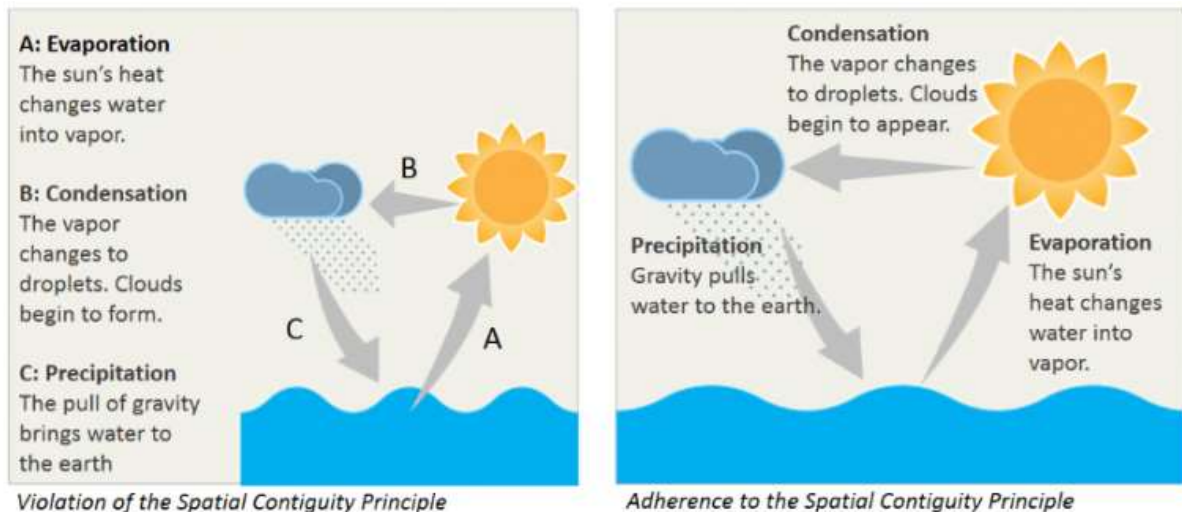


Figure 2.6: Two alternatives of representing information (water cycle example) [23].

However, the split attention effect can also occur in more occasions that spatial contiguity doesn't cover. Split attention effect, as the name suggests, is when a human's attention is divided between more than one set of information, undergoing unnecessary cognitive load [27], because of how the information is presented.

The occurrence of split attention effect is not limited to the non-spatial contiguity of information, it also occurs when information that should be connected is temporally separated. Another occurrence of this effect is when information is given to the user in more than one sensory channel. The split attention effect should have more impact when learners are subjected to, for example, text and speech, compared to just text or just speech.

Applying these principles to text-based code editors like eclipse, we can pinpoint some occurrences where split attention effect could be happening. For example, it is very common for two pieces of closely related code to be far apart in a text file, breaking spatial contiguity, and if scroll is required, information also becomes temporally spaced. This separation of information could be triggering a split attention effect, having an impact on developer's cognitive load, possibly increasing time required for code understanding.

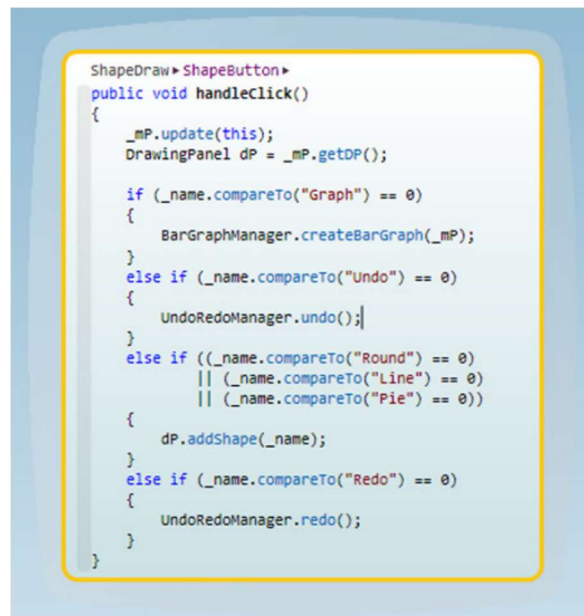
Because of the effects mentioned above, during the development of the proposed tool in this project, information juxtaposition should take into account spatial contiguity and split attention effect, making sure the placement and spacing of related content is appropriate.

2.3. Related Tools

This section presents some previously developed tools that have similar goals to those of PescaJ. These works are mostly editors that seek to display relevant information to the user without the need of superfluous navigation tasks.

2.3.1. CodeBubbles

Contemporary integrated development environments (IDEs) are file-based, which makes it difficult to visualize multiple sections of code simultaneously. CodeBubbles [2] is a code editor that seeks to fix this issue, providing the user with the possibility of viewing multiple editable fragments concurrently. In this project, the editable fragment's views are called *bubbles* [1], which can be of multiple types. Depending on the selected type, *bubbles* are able to display relevant information about functions, documentation, debugging, etc. Figure 2.7 shows an example of a text code *bubble*.



```
ShapeDraw ▶ ShapeButton ▶
public void handleClick()
{
    _mP.update(this);
    DrawingPanel dP = _mP.getDP();

    if (_name.compareTo("Graph") == 0)
    {
        BarGraphManager.createBarGraph(_mP);
    }
    else if (_name.compareTo("Undo") == 0)
    {
        UndoRedoManager.undo();
    }
    else if (( _name.compareTo("Round") == 0)
        || (_name.compareTo("Line") == 0)
        || (_name.compareTo("Pie") == 0))
    {
        dP.addShape(_name);
    }
    else if (_name.compareTo("Redo") == 0)
    {
        UndoRedoManager.redo();
    }
}
```

Figure 2.7: Text code *Bubble* [1].

Users can use a search bar to find classes and methods, which can be clicked to create a *bubble* in the environment, method calls in existing views can also be clicked to open a *bubble* with the respective method. *Bubbles* do not overlap, instead they push each other, moving organically, they also do not clip the contained content, using code reflow and elision. The *bubbles* inhabit a virtual space, where all the *bubbles* are visible at the same time. By having these multiple fragments concurrently visible, the tool seeks to aid in visual comparisons. The complete tool can be seen in Figure 2.8.

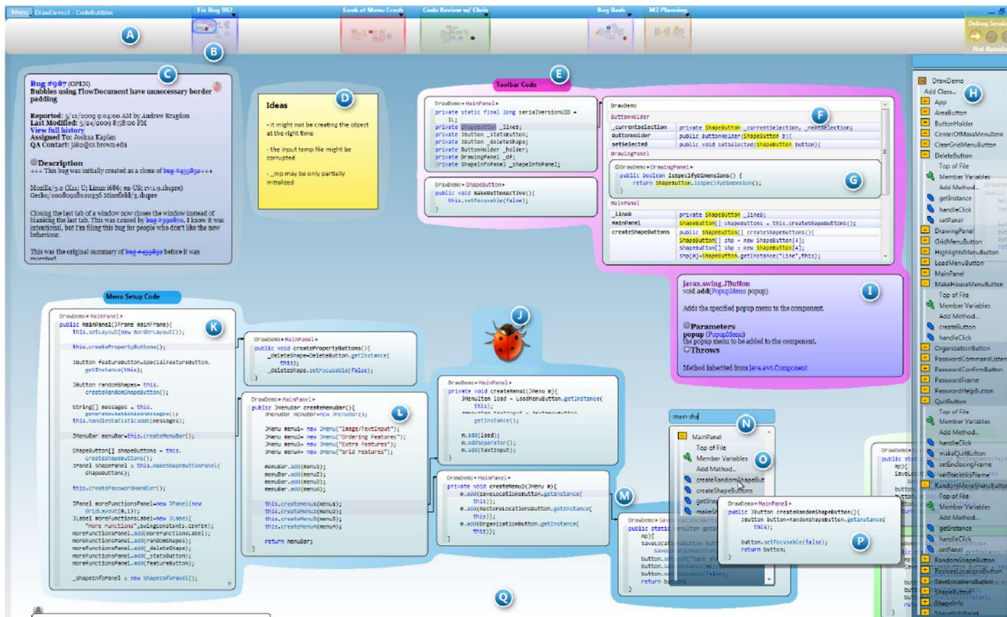


Figure 2.8: The CodeBubbles editor [2].

After comparing Eclipse with quantitative testing, it was determined that CodeBubbles significantly reduced the time of navigation required to understand code (68.6% reduction), as well as reducing the amount of navigation operations per minute (46.6% reduction). However, the navigation time only accounted for 34% of the improvement verified, they hypothesize that CodeBubbles not only helped developers reduce the time in navigation, but also lightened the load on their memory, speeding up tasks unrelated to navigation.

2.3.2. PatchWorks

PatchWorks [11, 12] are editors that seek to display more juxtaposed information, display relevant information side-by-side that does not require extra navigation to visualize. Instead of the classical tabbed editor approach, PatchWork tools use views called *patches*. These *patches* are views that display small editable fragments of the whole project. [11] is a code editor, with specialized *patches* to visualize code, as seen in Figure 2.9, while [12] is an editor for LabView, that works very closely to the former, but with *patches* to visualize LabView.

```

1  protected void fireBeginRedo()
2  {
3      for (BufferUndoListener listener: undoListeners)
4      {
5          try
6          {
7              listener.beginRedo(this);
8          }
9          catch(Throwable t)
10         {
11             Log.log(Log.ERROR,this,"Exception while sending buffer be
12             Log.log(Log.ERROR,this,t);
13         }
14     }
15 } //}}
16
17
18

```

Figure 2.9: A Patch from Patchworks [11]

The *patches* are arranged in a virtual *strip*, that is two *patches* high and infinitely long. The editor shows a close view of the *strip*, displaying up to six *patches* concurrently in [11], and four in [12], Figure 2.10 shows a representation of the virtual strip, as well as the patchworks editor with six visible patches.

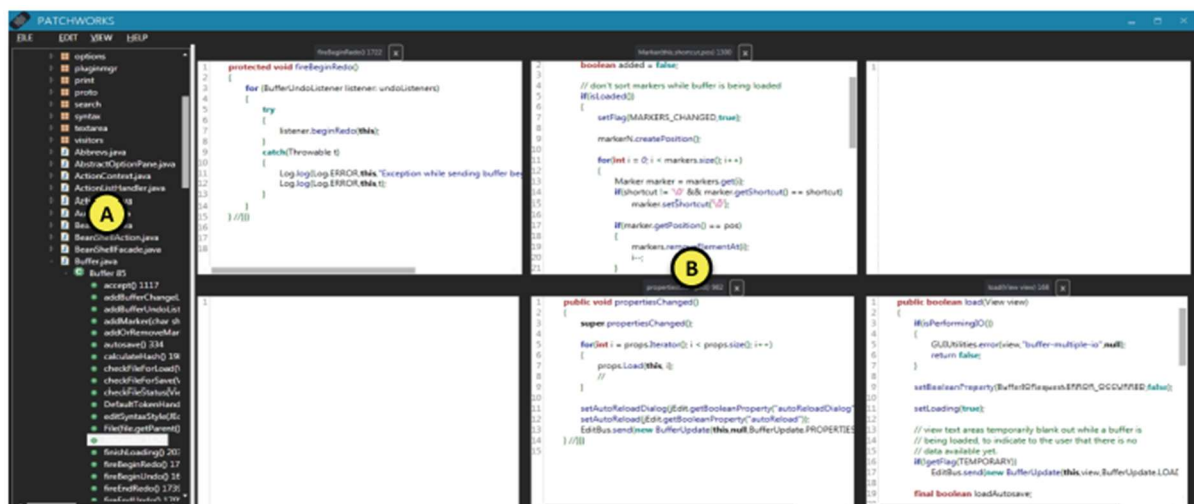
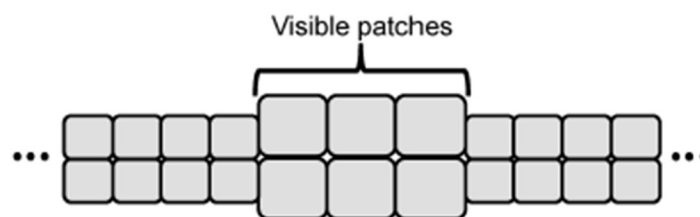


Figure 2.10: Representation of the *strip* of patches (top) and Patchworks Editor (bottom) [11].

The developer can navigate through the *strip* by sliding left and right, alternating the visible *patches*. PatchWorks also supports zoom out and a bird's eye view, so the developer can easily navigate through the *strip*, and find the needed patches, as shown in Figure 2.11.

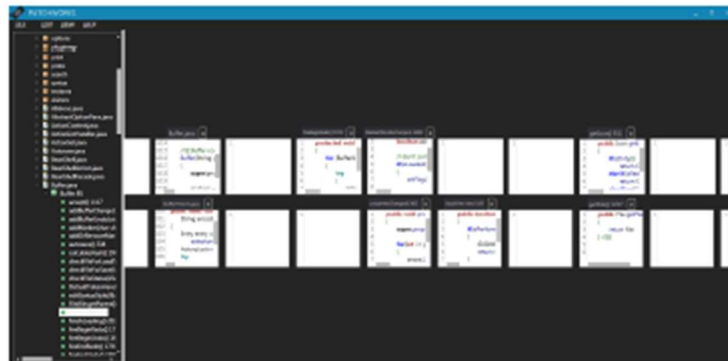


Figure 2.11: Patchworks with zoomed out *strip* [11].

A user study was conducted [11], where the navigation behaviors of students with at least 2 years of programming practice were compared to Eclipse and CodeBubbles [2]. Findings showed that, with PatchWorks and CodeBubbles, users spent less time doing navigation tasks. Patchworks users also showed less navigation mistakes when compared to the other two editors. Patchworks and Eclipse users spent less time rearranging code when compared with CodeBubbles.

2.3.3. Barista

Barista [Ko06] is an implementation framework that allows the display of relevant information embedded in the code editor. Text based code editors can only display graphs and images through HTML, and in most cases, these need to be opened in a separate window or dialog. In Barista, it is possible to place this relevant information next to the code, it also gives the possibility of more complex representations, like animation. Figure 2.12 shows a method in Barista, where the documentation has a visual representation of the method's behavior.

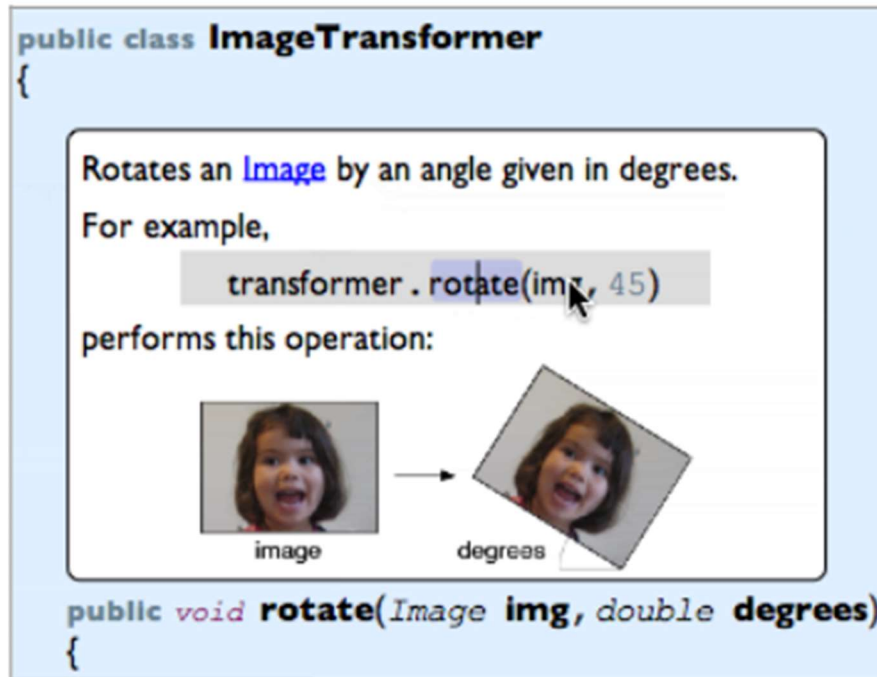


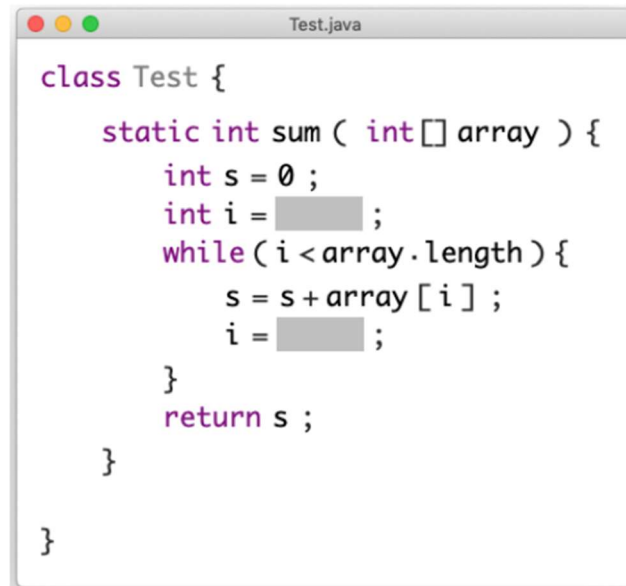
Figure 2.12: Method description with image in Barista [15].

Barista is a structured code editor that also allows unstructured writing. The framework's architecture consists of a Model-View-Controller pattern, where the model is the Abstract syntax tree of the code being edited, under this perspective, Barista could be considered a Projectional Editor, as earlier explained in Section 2.1.5.

2.3.4. Javardise

While learning programming, students are faced with two distinct challenges, essential difficulties that emerge from the inherent complexity of programming, like algorithms, and accidental difficulties, like the syntax of the language. Javardise [26] is a structured code editor for Java, that seeks to lighten the burden of accidental difficulties pertaining to programming language syntax in the learning process.

This tool automatically generates the structure of the code, meaning students don't have to worry about syntax exclusive artifacts like semicolons or curly brackets, being able to focus on essential difficulties, however, Javardise keeps the look and feel of editing in an unstructured text editor as much as possible, Figure 2.13 shows the Javardise code editor, the gray sections are input fields where the expressions can be written.

The image shows a window titled "Test.java" with a code editor. The code is as follows:

```
class Test {  
    static int sum ( int[] array ) {  
        int s = 0 ;  
        int i =  ;  
        while ( i < array . length ) {  
            s = s + array [ i ] ;  
            i =  ;  
        }  
        return s ;  
    }  
}
```

Figure 2.13: Javardise code editor [26].

Developing a code editor from scratch is a huge task and is outside the scope of the proposed tool in this dissertation, because of this, Javardise will play a key role in the development of the project, being imported and used as the widgets that enable the display and editing of code in PescaJ.

2.4 Behavior of Text-based Code Editors

To motivate the design of PescaJ, the behavior of other code editors was analyzed, this helped in detecting shortcomings in the designs and workflows of the editors that may have a negative impact on good navigation and code understanding. We use Eclipse as an illustration case for a set of features that can be found in other popular IDEs (e.g., IntelliJ, VS Code). The design of PescaJ will then take into account these findings, working on improving them.

As an example, we will use the source code of the *ArrayList* class, from the *java.util* package. This class comprises 1759 lines on code, a sizable amount of which are documentation (46%). The method *addAll* calls both intra-class methods *rangeCheckForAdd* and *grow*. These methods are not contiguous in the source files, *addAll* starts at line 730, *rangeCheckForAdd* at line 785 and *grow* at line 241.

In Eclipse, because the representation is tied to the source files, the methods cannot be shown on the same page simultaneously, due to the big distance of lines. The user has two main workflows for traversing between the methods. The first is, using only one window, setting the viewer at the desired method each time, this can be done with the use of shortcuts (e.g., *ctrl+click*), or using the search feature. However, the most friendly way is to open a split view for each method that needs to be visualized or edited, as seen in Figure 2.14.

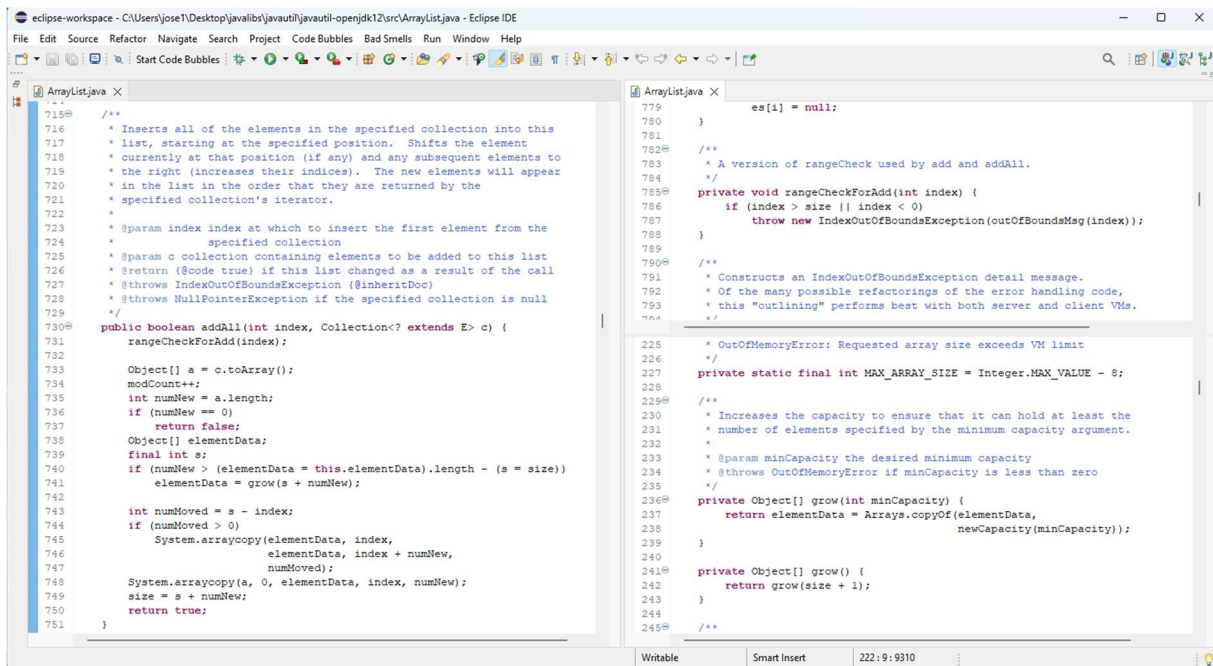


Figure 2.14: Methods of the same class displayed concurrently using multiple windows (Eclipse).

This work environment allows the user to concurrently visualize and edit all three methods. However, this solution does have its shortcomings, opening the split views is not seamless, as the feature is slightly hidden behind menus (*Windows* → *Editor* → *toggle Split editor*), and does require a moderate amount of navigation inputs, including navigating to the intended method after each view is opened. This feature is not very expandable either, in Figure 2.14 three methods are displayed, and the amount of clutter on the screen is already considerable. For an increased amount of code fragments, managing every fragment may become burdensome.

Unlike the previous example, while performing inter-class navigation, the number of lines between methods does not apply, since different classes are usually stored in separate source files (with some exceptions, such as non-public Java classes). Considering methods of distinct classes, such as `toArray` from the class `ArrayList`, which calls `copyOf` from the class `Arrays`, the user needs to consult multiple source files to navigate through these inter-class methods. This can be done, much like the last example, with the use of tabs. Tabs are a quick way to switch between source files without the introduction of clutter or confusion caused by multiple instances of Eclipse, they can also be displayed side-by-side.

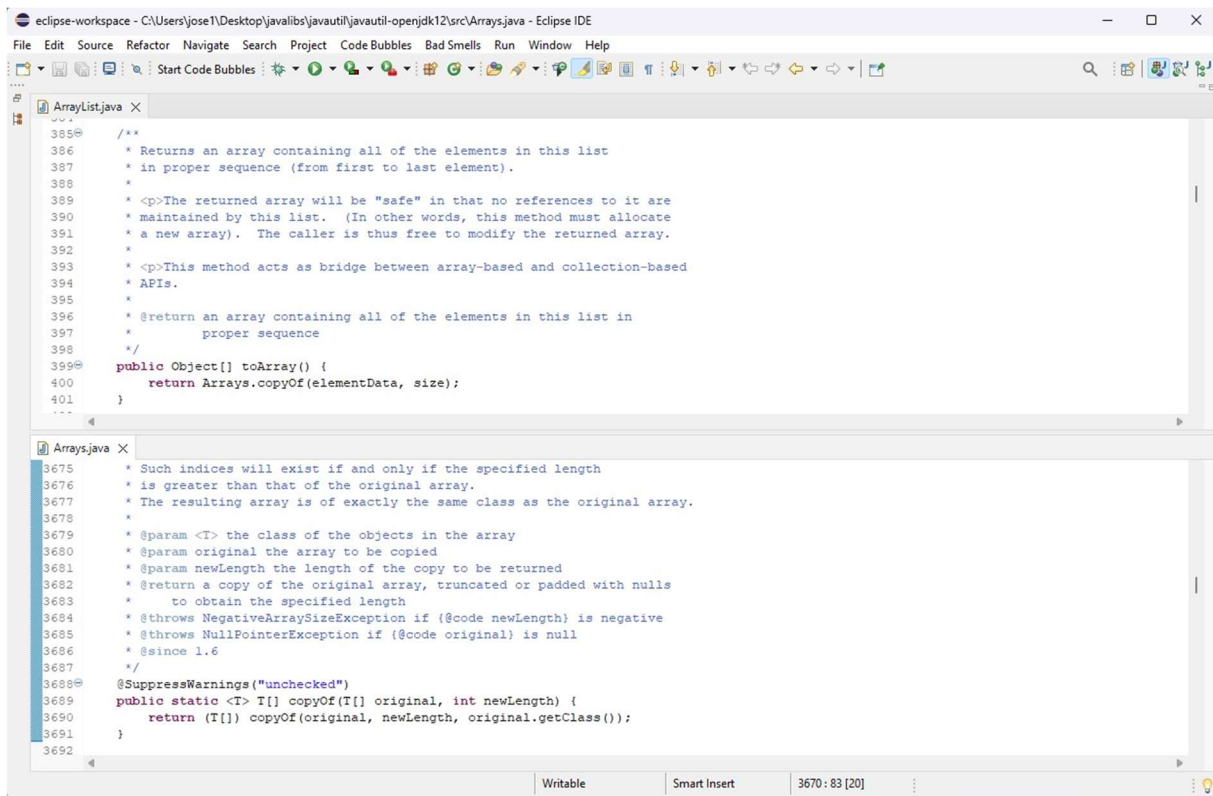


Figure 2.15: Two methods of distinct classes displayed concurrently using multiple tabs with Eclipse.

Figure 2.15 shows the use of tabs to display two methods of different classes concurrently, when compared to the previous example, there is a clear similarity between split views and tabs, however the later is working with distinct source files, while the split views are all feeding off the same file.

PescaJ's behavior should be inspired by the benefits given by the use of juxtaposed tabs and split views on classical editors like Eclipse. These benefits are the aggregation of closely related information (e.g., methods in a call chain), which reduces the need for navigation between code fragments, like methods scattered in a file, and enable concurrent visualization and editing. However, these features require a moderate amount of navigation effort to be usable, like the reorganization of tabs on the Eclipse window. In turn, PescaJ should allow the concurrent editing of methods in a more friendly way, allowing the creation of code fragments with seamless and intuitive user interaction. The resulting fragments should also imply less clutter and be easier to manage, when compared to Eclipse.

When it comes to documentation management, options are even more limited. While working with the source files, there is no easy way to aggregate relevant pieces of documentation together, the only feature that could help in this matter is *collapsing*. Users can choose to hide fragments of code or documentation, Eclipse even offers the option to *collapse* all code, which leaves only documentation visible, as seen in Figure 2.16.

```
ArrayList.java ×
278  /**
279     * Returns the number of elements in this list.
280     *
281     * @return the number of elements in this list
282     */
283  public int size() {}
286
287  /**
288     * Returns {@code true} if this list contains no elements.
289     *
290     * @return {@code true} if this list contains no elements
291     */
292  public boolean isEmpty() {}
295
296  /**
297     * Returns {@code true} if this list contains the specified element.
298     * More formally, returns {@code true} if and only if this list contains
299     * at least one element {@code e} such that
300     * {@code Objects.equals(o, e)}.
301     *
302     * @param o element whose presence in this list is to be tested
303     * @return {@code true} if this list contains the specified element
304     */
305  public boolean contains(Object o) {}
308
309  /**
310     * Returns the index of the first occurrence of the specified element
311     * in this list, or -1 if this list does not contain the element.
312     * More formally, returns the lowest index {@code i} such that
313     * {@code Objects.equals(o, get(i))},
314     * or -1 if there is no such index.
315     */
316  public int indexOf(Object o) {}
319
320  int indexOfRange(Object o, int start, int end) {}
337
338  /**
339     * Returns the index of the last occurrence of the specified element
340     * in this list, or -1 if this list does not contain the element.
341     * More formally, returns the highest index {@code i} such that
342     * {@code Objects.equals(o, get(i))},
343     * or -1 if there is no such index.
```

Figure 2.16: Source code file with collapsed code, focusing of the display of documentation (Eclipse).

The only way to properly aggregate documentation is with the generated HTML files, this is especially useful to visualize documentation for class and package. However, these generated files are not editable, which makes their navigation and concurrent editing difficult.

As mentioned in Section 1.1, the task of writing code and its documentation may be considered two separate activities. In this light, documentation and code may be considered *tangled*, seeing as they are both stored in source files. This is visible in Eclipse, where lines of documentation are essentially clutter when coding, and vice-versa.

Empirical Study of Method Dependencies and Documentation

One of PescaJ's goals is to more accurately display code structure and the interaction between fragments, such as displaying the call graph of methods. Because of this, it is of great importance to pinpoint the most common code structure trends, enabling the development of an interface capable of handling them.

The goal of this small study was to better understand common patterns in code structure, answering the research question Q1 mentioned in Section 1.3, and use these findings in the development of the tool. We chose a small collection of metrics that we think will prove useful in characterizing the most common call graphs in code. Aside from code structure, the quantity of documentation was also analyzed, to motivate the documentation facet of PescaJ. The study was conducted on the Qualitas Corpus curated Java project collection, as well as the *java.util* package from Java.

In this chapter, we will describe and characterize the collection *Qualitas Corpus* and the *java.util* package as a sample for the study, explain how the study was conducted and the results gathered. Finally, we will analyze the results and take away some conclusions that will be useful for the developments of the proof-of-concept tool.

3.1. Metrics

In this section we will describe the methodology used for gathering the metrics, explaining how the developed script works. We will also describe each metric, and the details involved in their definition.

To obtain the desired metrics for further analysis, a script was developed to scan the code samples. The script uses JavaParser [3] to load the code into memory and traverse the methods in the code, gathering important values in the process. After each project is scanned, the values of the desired metric are inserted into a Comma-Separated-Value (CSV) file and saved for later processing.

In a first approach, all the projects in *Qualitas Corpus* were being scanned in succession and the writing would be done at the end of the analysis. However, the sheer size of the project collection caused the script to run out of memory, proving this approach to be unviable. To circumvent the memory issue, projects were scanned in batches, and metrics for each project were saved in separate files. After every batch was scanned by the script, Excel was used to import all the CSV files and compile the results into a single file.

Some projects of the *Qualitas Corpus* proved to be too large to be scanned by the script, as memory was exhausted, out of the 112 systems 96 were successfully analyzed, this means the script was not able to handle 18 of the projects. Since the number of unparsed projects was relatively low compared to the sample size (16% of projects of the total *Qualitas Corpus* and close to 19% of the parsed projects), we opted to not further optimize the script looking for a solution, projects that proved too big to be analyzed were simply discarded, as previously mentioned, this resulted in a total of 96 projects from *Qualitas Corpus*, which we consider an acceptable number for the purpose of this study.

At this point we made the decision of only taking into account method calls that point to a method inside the project being analyzed. This means imported methods and methods of the Java language (e.g., `System.out.println()`) aren't counted in this analysis. This is because the tool will be applied to a project and only code of that project will be able to be visualized and edited. We also only considered methods with the public visibility as an entry point for the scanner, this is because users will most likely use public methods as an entry point for code exploration in PescaJ. However, it is important to remember that the restriction of being public is only applied as an entry point, this means a public method that calls two public methods and three private methods will still have a number of methods called of five. We believe these restrictions make sense in the context of PescaJ, resulting in useful metrics for the development of the tool, however it is relevant to keep in mind the results are not fully representative of the code analyzed, with the context of the metrics being of utmost importance in the understanding of their values.

The chosen metrics were the following:

- **Method calls:** This metric indicates how many unique methods are called by a given public method, it represents the width of the average call graph starting at public fragments. It is subdivided by calls to methods of the same class and methods outside the class, resulting in the following metrics:
 - Number of intra-class method calls
 - Number of inter-class method calls
 - Total number of method calls
- **Call depth:** This metric indicates the size of the biggest chain of calls starting at a given public method, it represents how deep the average call graph that starts at public methods is. The metric is also measured for methods only inside of the same class, resulting in the following metrics:
 - Maximum intra-class call depth
 - Max call depth

- **Lines of Documentation:** For this metric, the lines of documentation were counted for each file, as well as the total lines of the file, it will be used to motivate the usefulness of the dedicated documentation views.

3.2. Qualitas Corpus

Measuring code is an essential step in understanding its structure, not only to improve code writing (e.g., using code smells to detect badly written code), but also to create new ways to interact with it. These types of studies have been developed for decades, and across multiple languages. The necessity of studies like these is visible, with empirical studies done in languages like FORTRAN [13], COBOL [4], Unix [21], C++ [10, 5], Java [30, 9], among many others.

However, empirical studies of code are difficult to execute, even with a vast quantity of open-source code to analyze, the task of gathering the individual systems and guaranteeing their quality is considerable.

Looking to fill this need, E. Tempera et al developed *Qualitas Corpus* [28], a curated collection of open-source Java systems. There are multiple versions of *Qualitas Corpus*, the most basic ones contain the most recent version of the systems, while there are others that keep a history of versions and branches made in active development. For the purpose of this study, we found the version of *Qualitas Corpus* that only contains the most updated version of the systems to be most appropriate. The version *20130901r* of the *Qualitas Corpus* was used, containing 112 systems. As previously mentioned, not every project was parsed due to memory constraints. In total, 96 projects were scanned, summing up 437.796 methods in total.

The median value for the number of methods in a project was 2.980, with the average value of 4.560. Figure 3.1 shows a plot of the distribution of projects by the number of methods they contain. While there are some outliers that raise the average, we can see the majority of projects are smaller, with half the projects having less than 3.000 methods.

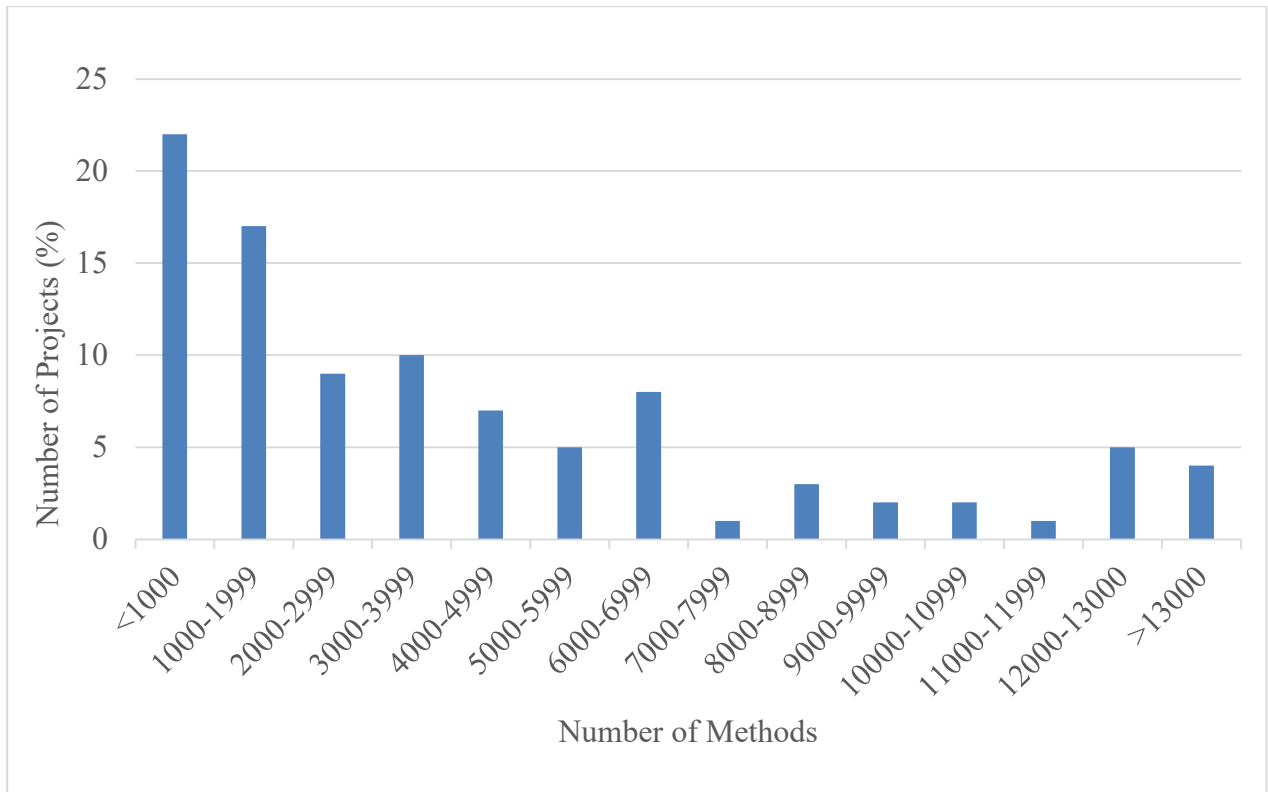


Figure 3.1: Distribution of number of projects (%) in *Qualitas Corpus* per number of methods.

3.3. Package *java.util*

The *java.util* is a package of Java's standard libraries, it is responsible for some of the most important functions and data structures for Java development, such as the collections framework, date and time facilities and other utility classes (source code of OpenJDK 12+32)². The package is open source and consists of 119 Java source files.

The Java package *java.util* was scanned, showing results for 85 files. The discrepancy between this number and the total number of files is expected, seeing as some files contain things like interfaces (e.g., *Observer.java*) or classes which contain only private or protected methods (e.g., *ComparableTimSort.java*). The resulting metrics originated from 2.805 methods, with classes like *Collection* taking a huge slice of this number, with 610 methods.

² <https://github.com/openjdk/>

3.4. Results

This section will be focused on the resulting metrics from the study, the values of each metric will be shown for both *Qualitas Corpus* and the *java.util* package. An analysis of the values will be done and, finally, some useful conclusions to take away for the development of PescaJ.

For each metric a number of tables will be shown, in these tables, values are highlighted in blue, this represents the values that make up 99% of the occurrences. For example, the first table, Table 3.1, has a blue highlight up to the value three, this means 99% of cases call up to three methods, leaving higher values with a residual presence.

3.4.1. Method Calls

Tables 3.1 and 3.2 show the distribution of methods by number of intra-class method calls. The majority of methods (83,32% for *Qualitas Corpus* and 69,30% for *java.util*) do not call any methods of the same class, this value was expected and will be a trend among other metrics. Considering we are analyzing Java projects, a large number of methods scanned will be getters and setters, padding the number of methods calling zero methods.

On *Qualitas Corpus*, methods that call up to three intra-class methods make up 99% of the sample (highlighted in blue in Table 3.1), with diminishing presence for more methods called. On the *java.util* package, the scenario is slightly different, with more presence of methods that call a bigger number of intra-class methods, however 99% of occurrences still happen at a reasonable number, with 5 methods called.

Number of Intra-Class Method Calls	Count	Count %
0	364780	83,32%
1	54749	12,51%
2	11046	2,52%
3	3391	0,77%
4	1735	0,40%
5	844	0,19%
6	405	0,09%
7	226	0,05%
8	164	0,04%
9	110	0,03%
10+	346	0,08%
Grand Total	437796	100%

Table 3.1: Distribution of methods by number of intra-class method calls for *Qualitas Corpus*.

Number of Intra-Class Method Calls	Count	Count %
0	1944	69,30%
1	593	21,14%
2	163	5,81%
3	43	1,53%
4	20	0,71%
5	16	0,57%
6	15	0,53%
7	5	0,18%
8	3	0,11%
10+	3	0,11%
Grand Total	2805	100%

Table 3.2: Distribution of methods by number of intra-class method calls for the *java.util* package.

For inter-class methods called, as we can see in Table 3.3 and 3.4, almost every *Qualitas Corpus* method (97,84%) and the big majority in *java.util* (86,31%) does not call a method of other classes. It is important to remember this metric only takes into account methods of other classes inside the project, methods imported from outside sources or Java libraries do not count towards this value. The *java.util* package shows a small presence of methods that make one or two inter-class method calls, this is to be expected since it is a Java library, it contains a lot of classes that are built using other utilities also from *java.util*, such as *ArrayList* being built using *Arrays*.

Number of Inter-Class Method Calls	Count	Count %
0	428332	97,84%
1	5221	1,19%
2	1984	0,45%
3	822	0,19%
4	447	0,10%
5	278	0,06%
6	248	0,06%
7	111	0,03%
8	82	0,02%
9	45	0,01%
10+	226	0,05%
Grand Total	437796	100%

Table 3.3: Distribution of methods by number of inter-class method calls for *Qualitas Corpus*.

Number of Inter-Class Method Calls	Count	Count %
0	2421	86,31%
1	321	11,44%
2	51	1,82%
3	5	0,18%
4	4	0,14%
5	2	0,07%
10+	1	0,04%
Grand Total	2805	100%

Table 3.4: Distribution of methods by number of inter-class method calls for the *java.util* package.

If we consider both inter- and intra-class method calls, we get the distributions shown in Table 3.5 and 3.6. The scenario is similar to the last cases, however in this case, 99% of methods call up to four methods for the *Qualitas Corpus* (highlighted in blue in the Tables) and five for *java.util*. There is a notable difference between the two cases, *java.util* shows a higher number of method calls overall when compared to *Qualitas Corpus*, with only 58,54% of methods making no inside project method call when compared to 81,75% of *Qualitas Corpus*.

Total Number of Method Calls	Count	Count %
0	357904	81,75%
1	57000	13,02%
2	12929	2,95%
3	4445	1,02%
4	2196	0,50%
5	1132	0,26%
6	702	0,16%
7	367	0,08%
8	286	0,07%
9	169	0,04%
10+	666	0,15%
Grand Total	437796	100%

Table 3.5: Distribution of methods by total number of method calls for *Qualitas Corpus*.

Total Number of Method Calls	Count	Count %
0	1642	58,54%
1	783	27,91%
2	251	8,95%
3	52	1,85%
4	30	1,07%
5	19	0,68%
6	14	0,50%
7	4	0,14%
8	4	0,14%
9	2	0,07%
10+	4	0,14%
Grand Total	2805	100%

Table 3.6: Distribution of methods by total number of method calls for the *java.util* package.

Considering these results, it is safe to say the vast majority of methods have a reasonable number of method calls, with more intra-class method calls than inter-class. In every case, the biggest slice belongs to methods with zero calls, as previously mentioned, that is caused by the large quantity of accessory methods present in the Java language, as well as methods that use external code, which are not considered in this study.

The value of called methods is a good indicator of the width of the call graph, and the developed tool should have the capacity to juxtapose appropriately up to four called methods. PescaJ should also be able to juxtaposed methods of distinct classes, however the inter-class method call metric was relatively low, meaning the number of classes that are required to be displayed concurrently is feasibly low. While methods that call large amounts of methods do exist, they are a residual amount, the tool should not be designed to accommodate these edge cases.

3.4.2. Call Depth

Table 3.7 and 3.8 show the distribution of methods by maximum intra-class call depth. Methods with a maximum call depth of up to three represent 99% of the *Qualitas Corpus* (highlighted in blue), while in *java.util* this number is elevated to five.

Intra-Class Max Call Depth	Count	Count %
0	364780	83,32%
1	51319	11,72%
2	13757	3,14%
3	4485	1,02%
4	1396	0,32%
5	751	0,17%
6	239	0,05%
7	96	0,02%
8	56	0,01%
9	22	0,01%
10+	895	0,20%
Grand Total	437796	100%

Table 3.7: Distribution of methods by intra-class maximum call depth for *Qualitas Corpus*.

Intra-Class Max Call Depth	Count	Count %
0	1944	69,30%
1	487	17,36%
2	199	7,09%
3	90	3,21%
4	47	1,68%
5	22	0,78%
6	8	0,29%
7	5	0,18%
8	2	0,07%
9	1	0,04%
Grand Total	2805	100%

Table 3.8: Distribution of methods by intra-class maximum call depth for the *java.util* package.

Table 3.9 and 3.10 shows the overall maximum call depth of methods, much like before, 99% of methods have a maximum call depth of up to four on *Qualitas Corpus* and five on *java.util*. However, it is relevant to point out a small discrepancy in the number of methods with zero call depth.

Max Call Depth	Count	Count %
0	357889	81,75%
1	53935	12,32%
2	14997	3,43%
3	5600	1,28%
4	2057	0,47%
5	1234	0,28%
6	484	0,11%
7	263	0,06%
8	141	0,03%
9	60	0,01%
10+	1136	0,26%
Grand Total	437796	100%

Table 3.9: Distribution of methods by maximum call depth for the *Qualitas Corpus*.

Max Call Depth	Count	Count %
0	1642	58,54%
1	628	22,39%
2	258	9,20%
3	136	4,85%
4	82	2,92%
5	32	1,14%
6	11	0,39%
7	6	0,21%
8	5	0,18%
9	5	0,18%
Grand Total	2805	100%

Table 3.10: Distribution of methods by maximum call depth for the *java.util* package.

It is relevant to notice the number of methods with zero call depth is the same as the number of methods that call zero methods shown in last section (81,75% on *Qualitas Corpus*), this value should always be the same, because if a method calls no other methods, it always has a call depth of zero. The absolute number shown in Table 3.9 (357.889), should be exactly the same as methods with zero called method (357.904) in Table 3.5, however the number has a difference of 15 methods. This difference is attributed to a small issue with the parser, in some cases, projects that implemented methods like *run()*, were counted for the *method calls*, but were not counted for *maximum call depth*. Apart from this small discrepancy, the numbers match up as expected.

The results imply that the vast majority of methods have a reasonably small depth of call graph, which means the developed tool should be able to comfortably display up to four methods in depth. The *java.util* package showed a deeper call graph on average, however the vast majority of methods still only show a maximum depth of four, with only 2,1% with a depth of five or more.

3.4.3. Lines of Documentation

In total, *Qualitas Corpus* contained 3.647.959 lines of documentation out of 14.113.895 total lines, with documentation making up 25,85% of lines in the source files of the *Qualitas Corpus* projects. Figure 3.2 shows the distribution of these files by percentage of documentation, showing a steady decline of the number of files above 40-50% of lines of documentation.

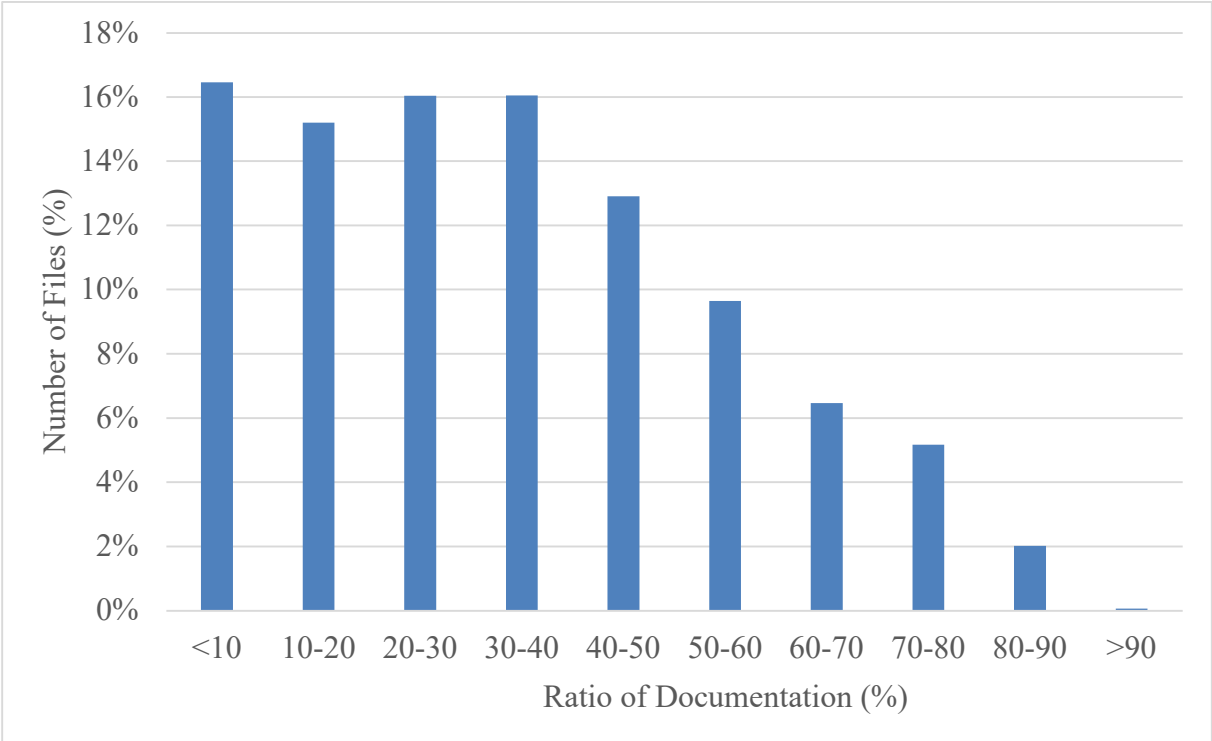


Figure 3.2: Distribution of files per amount of documentation lines(%) for *Qualitas Corpus*.

On the other hand, the *java.util* package contains a lot more documentation, with 51.002 lines of documentation out of 103.520 total lines, with documentation making up 49,26% of the total lines in the source code. Figure 3.3 shows that a big slice (more than 25%) of files are made up of 60-70% of lines of documentation, with very few files containing less than 30% of lines of documentation.

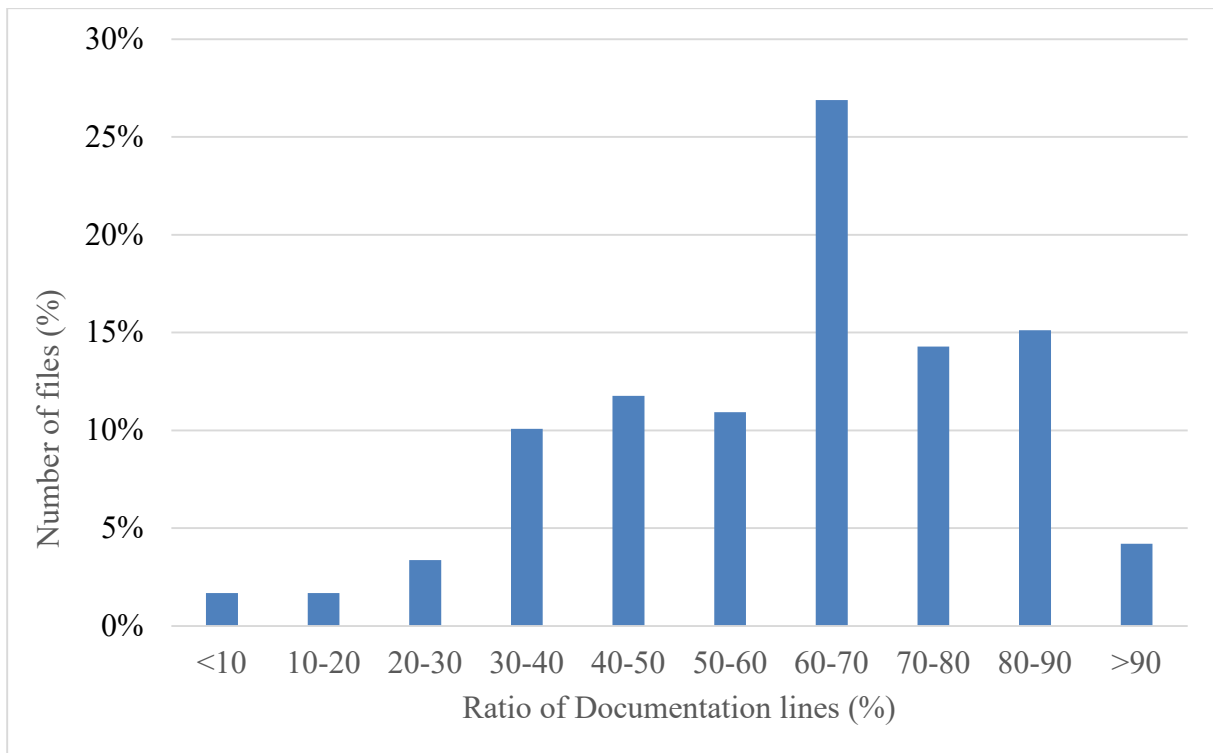


Figure 3.3: Distribution of files per amount of documentation lines for the *java.util* package.

The *java.util* package shows a staggering amount of documentation, however it is part of Java's standard library, and as such, it is extremely well documented. This level of documentation cannot be expected from the average project, however even the values obtained from *Qualitas Corpus* show a considerable number of lines of documentation. Since we consider the development of code and the writing of its documentation to be two separate tasks, or *concerns*, PescaJ should employ mechanisms to separate the code and its documentation, reducing clutter and separating *concerns*, reducing *tangling*, justified by the considerable amount of documentation present in source code files.

CHAPTER 4.

PescaJ

Using the information collected in the study of Chapter 3, and inspired by the strengths and weaknesses of typical code editors, explored in Section 2.4, and others like CodeBubbles [2] and Patchworks [11], we designed PescaJ. This chapter describes the main architectural aspects of PescaJ. Afterwards, we will shed a light into the main design choices of PescaJ, showcasing its features with practical examples. Finally, some technical implementation choices made in development will be discussed.

4.1. MVC architecture with overlapping views

The backbone of PescaJ is its architecture, which enables the creation of views that can aggregate fragments of code, constructed by interaction with a model, the Abstract Syntax Tree (AST) of the project being edited. In this section we will highlight the main features of this architecture, and the practical impact it entails.

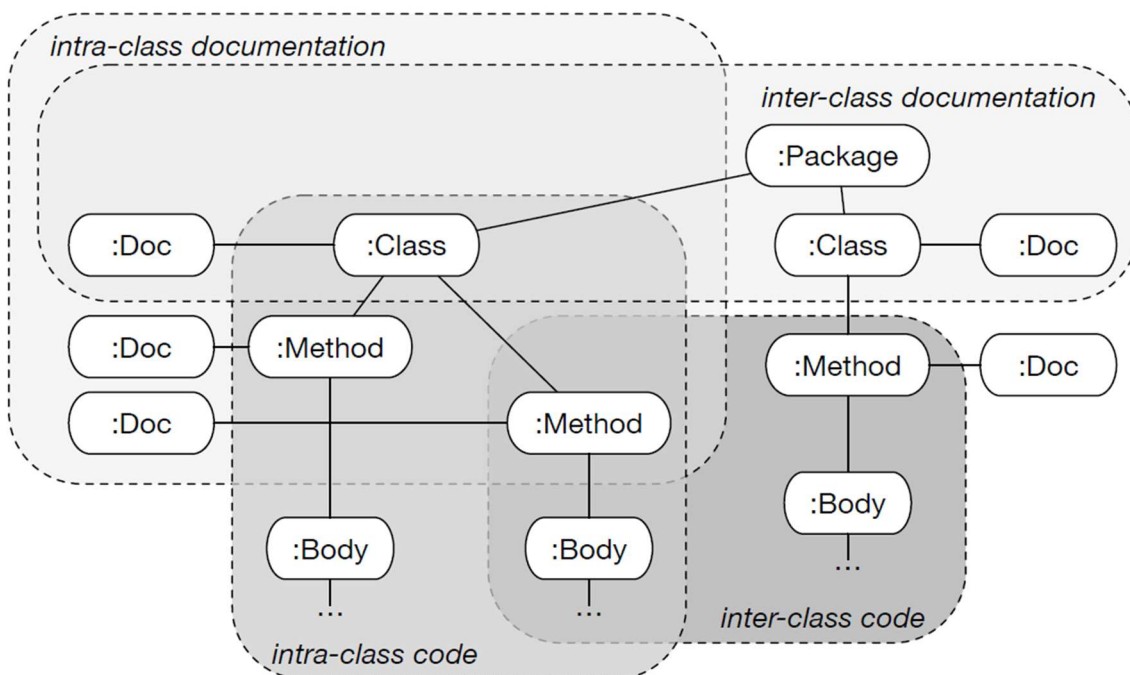


Figure 4.1: MVC architecture using overlapping views over fragments of the ASTs.

Figure 4.1 illustrates the MVC architecture used in PescaJ, at the core of the model we can see connected nodes, this represents the (AST) of the project that is being edited. Surrounding the AST are the views (dotted boxes) that interact with each node, these enable visualization and editing of the AST by the user. As represented in Figure 4.1, the views overlap, meaning each node may interact with more than one view, being able to be edited by each one concurrently and giving the possibility of distinct representation. In practice, this means code fragments may be displayed in multiple views, without being bound to their source code representation, this will enable aggregation and juxtaposition of normally scattered code or documentation fragments.

Another important aspect is that views do not cover the entirety of the AST, this means each view does not need to interact with all of the existing nodes, for example, the *intra-class documentation view* only interacts with the documentation nodes, ignoring the body nodes. In the same way, the code views, interact with the body nodes but disregard the documentation nodes. This design choice makes it possible to enact separation of concerns, with each view being able to display only the relevant nodes in a given context. The removal of irrelevant information also contributes to the reduction of clutter. The features enabled by this type of architecture closely follow the goals set for PescaJ, whose resulting design is detailed in the next section.

4.2. Design

PescaJ was built as a standalone editor, it was designed to use well known navigation elements and paradigms used in conventional editors, to reduce friction for experienced developers. Figure 4.2 shows a screenshot of PescaJ in action, using a real example with three methods of the class *ArrayList* (Java libraries), annotated with the line distances in the source code file.

The section at the top of the editor is a bar for drop-down buttons. This is where the workflow of PescaJ starts, using the *Project* button, a file chooser pop-up menu is displayed, where the user can browse and select the *src* directory of the Java project to be loaded.

After parsing the project for a brief moment, PescaJ populates the area to the left of the window, the package explorer, where methods, classes and packages loaded from the chosen project are placed in alphabetical order. This element was closely inspired by the *Outline* view in Eclipse, serving a similar purpose. This component is equipped with a search bar, which allows the user to find specific methods or classes, using a search by string, or using a list of filters, such as visibility modifiers or documentation status. By selecting methods in this window, the editor will create a view containing this method and place it at the workspace, the section at the center of the editor.

Due to limited screen space, it is difficult to display all information simultaneously while keeping the font size reasonable. To circumvent this issue, *Pescal* employs *workspaces*, which are similar to the concept of *working set* of CodeBubbles [2]. These are canvases that can hold the different views provided by the tool. They enable users to create clusters of information readily available at all times, for example, a developer may use a *workspace* to edit the code of a class and another to visualize and edit its documentation. Alternatively, a developer could use different *workspaces* for working on distinct features in parallel without scrolling within or switching between files. The behavior of *workspaces* holding code and *workspaces* holding documentation is distinct, both versions will be analyzed and explained in the following section.

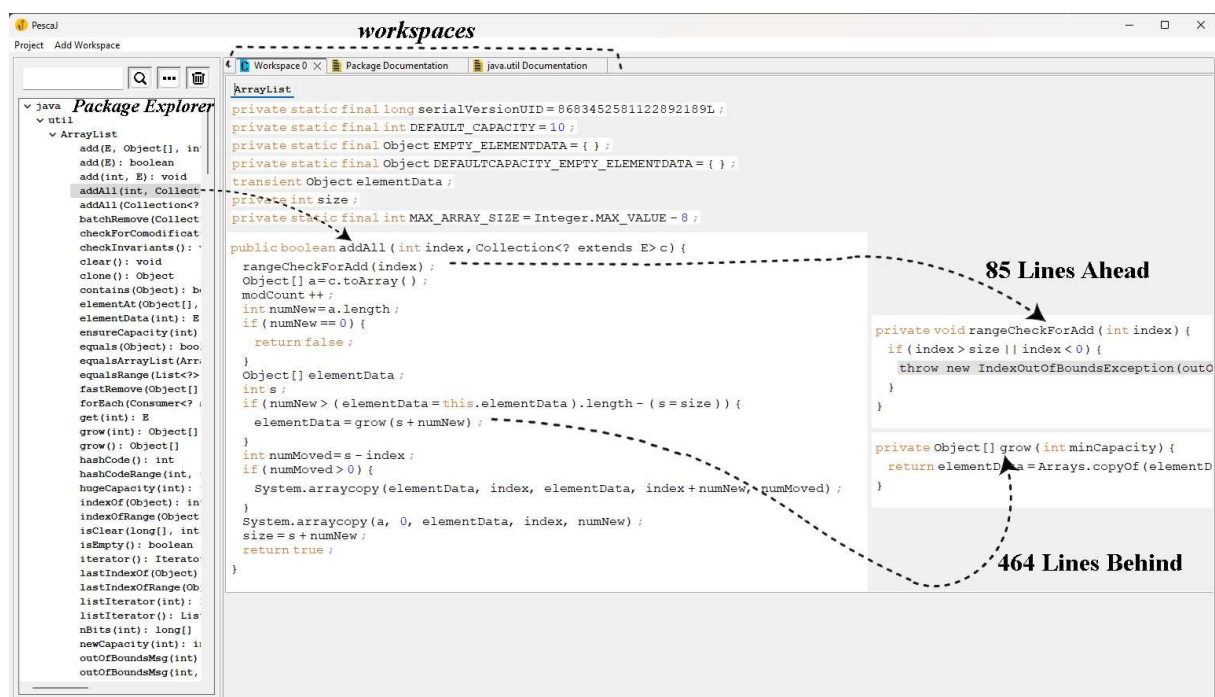


Figure 4.2: Pescal displaying an intra-class view illustrated with a fragment of the *ArrayList* class (Java libraries), juxtaposing methods according to call dependencies.

4.2.1. Code views

The code views are the interface elements where the user can edit and visualize code. These elements can contain methods and classes, and are placed in a *workspace*. When creating a *workspace*, the user may include views therein by dragging classes and methods from the package explorer. Views may be expanded with new elements also from interaction with the code of their elements. When a code view is created, its placement on the *workspace* depends on how the latter is already populated, and the code view's code fragment relation to other fragments.

Code *workspaces* are effectively organized in columns, the number of columns of the workspace is tied to the depth of the call graph. Figure 4.3 shows an example of three deep call graph, where the method *One* calls *AUXOne* which calls *AUXTwo*. Creating the chain: *One*→*AUXOne*→*AUXTwo*. Method *One* is not called by any previously placed method, so it is placed in column one. The method *AUXOne* is called by the previously placed method *One*, because of this, *AUXOne* should be placed to its right. If there is already a column in that position, *AUXOne* is placed there, otherwise, a new column is created and the placement is then completed, resulting in column two. *AUXTwo* is placed in Column three by the same logic. The method *Setter* does not interact with any of the previously mentioned methods, so it is placed at the left, in column 1.

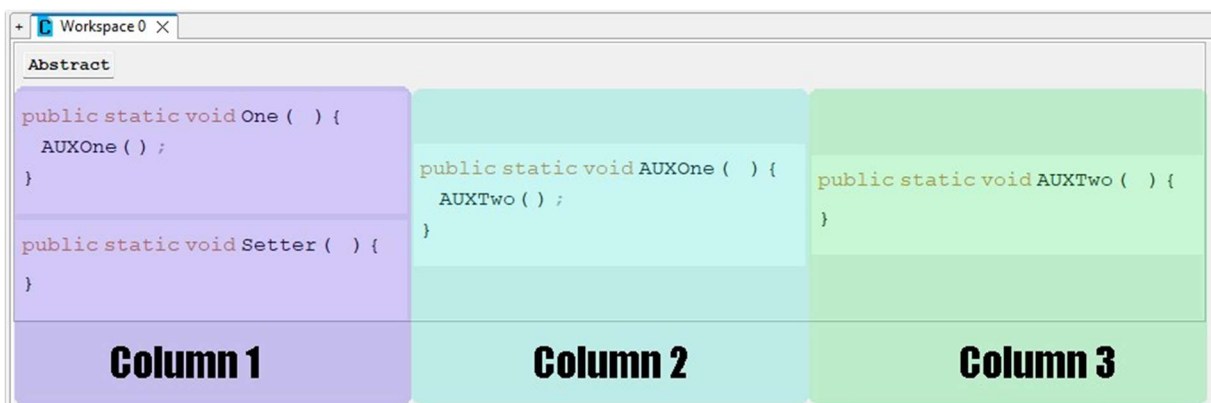


Figure 4.3: Pescal displaying three methods in a call chain with an overlapped scheme of the columns used for organizing.

This feature is not limited to the code view being added, in the last example, code fragments were opened according to their call order, however the proper order is also enforced when methods are opened out of order. In Figure 4.4, the methods *One* and *AUXTwo* were opened, since both are not called by anything in the workspace, they are both placed in the first column. When the method *AUXOne* is opened, the call chain is completed, more columns are created, and the workspace is rearranged to illustrate the full chain.

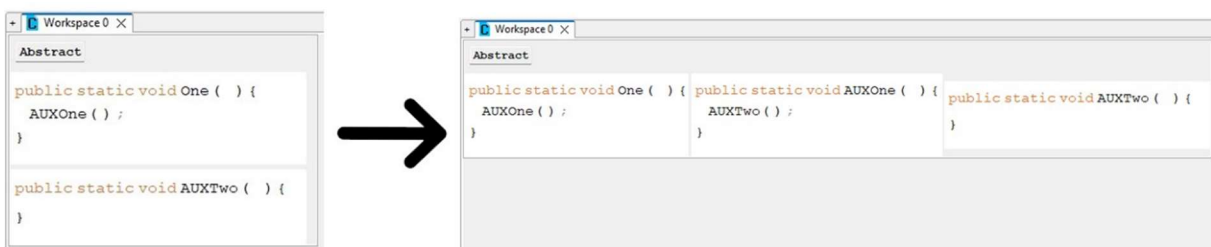


Figure 4.4: The organization of methods before (left) and after (right) a call chain is complete.

This sort of ordering is not limited to columns, vertical organization is also done to preserve the order of calls. This means that a method that is called first will always be placed on top of others that may be called later. The placement system is used to not only organize methods inside a class, but to also organize classes among each other. This automatic placement system ensures code fragments are as close as possible to their calls without the need of reorganization by the user, this proximity should facilitate code reading, lowering impact of the user's memory load, according to the principle of visual contiguity [27]. The close proximity of these fragments may also benefit to a lower amount of navigation errors, following Fitt's Law [20].

Code views are not limited to calls between methods of the same class, it can also represent dependencies between inter-class methods, creating multiple class widgets that are placed by the same logic seen in the last example. For example, as illustrated in Figure 4.5, by clicking the method call inside *toArray* of the class *ArrayList*, since the called method is from another class, a class widget is created for the class *Arrays*, and placed next to the one corresponding to the calling method. In typical code editors, these two fragments are found in two separate text files.

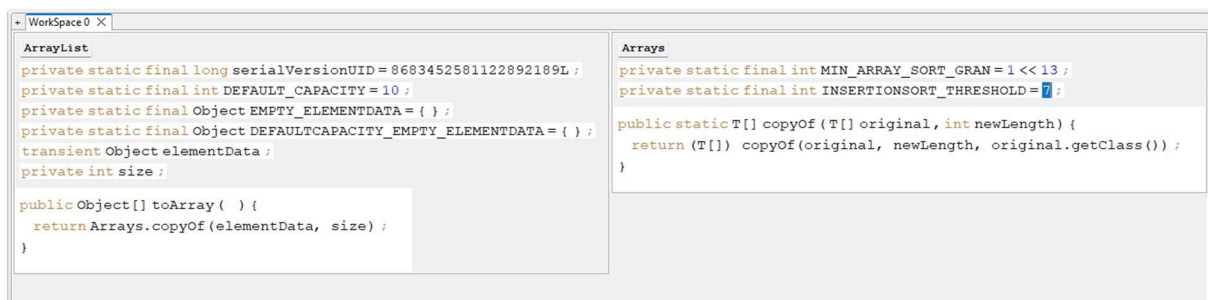


Figure 4.5: An inter-class view illustrated with a fragment of the classes *ArrayList* and *Arrays* (Java libraries), where the method *toArray* depends on method *copyOf*, which are juxtaposed.

4.2.2. Documentation views

The documentation views are responsible for the aggregation of documentation elements for various code fragments. While using these views, the user can visualize and edit the documentation of chosen code fragments, making tasks like comparing documentation style and coherence easier. Pascal provides four documentation views, which resemble the decomposition of the documents generated by *Javadoc*. These three views are essentially a list of members with their corresponding documentation header, being able to display and edit the documentation fragments of packages (Figure 4.6), classes (Figure 4.7), and methods (Figure 4.8).

In these list-style views, there are warnings that provide an indication of non-documented members contained therein. For example, while viewing the list of classes, those which have public non-documented methods contain a warning, allowing the user to know what classes need to be documented, without browsing each one individually. When hovering the warning additional information is presented, such as the number of public non-documented methods seen in Figure 4.6. We also allow filters to, for instance, only show the public members (as they are more critical for API documentation). The aim of these sorts of features is to embody an environment focused and specialized for documentation concerns (as opposed to the conventional code-embedded form).

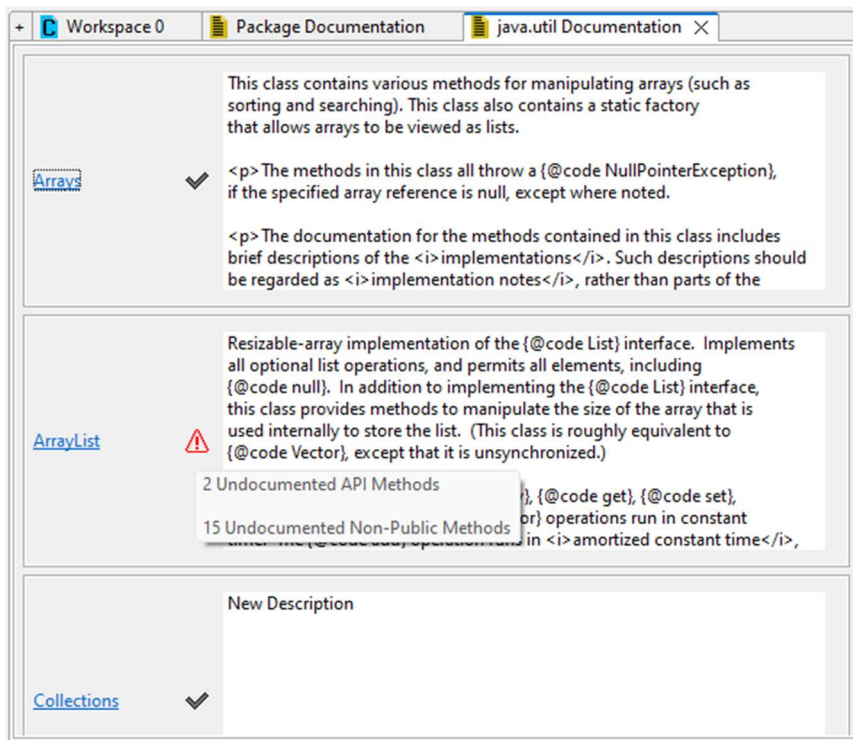


Figure 4.6: View that aggregates the documentation of classes in a package. The warning shows that the class *ArrayList* contains public undocumented methods.

In the documentation views the user may click a link in the declaration to navigate to that component's documentation. For example, by clicking the *ArrayList* link seen in Figure 4.6, the user would be presented with that class's documentation view, as well as the documentation for the methods in the *ArrayList* class, as shown in Figure 4.7.

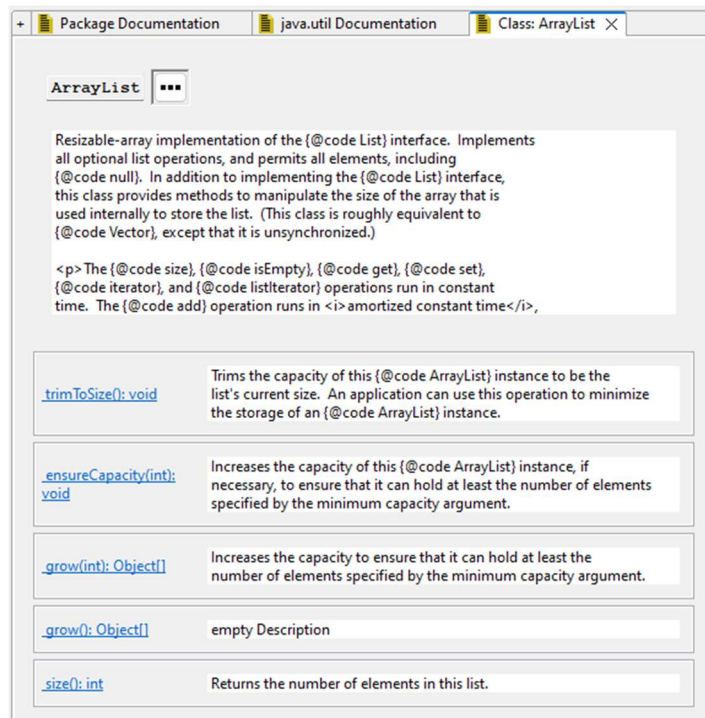


Figure 4.7: View that aggregates the documentation of the methods in a class, showing a documentation header for each method.

While in the class's documentation, only the headers of each method's documentation are displayed, to preserve screen space and reduce clutter. The user can expand these methods into a detailed view that shows the complete documentation of the method (see Figure 4.8), including the expanded description and tags. The tags are clustered together by type to reinforce visual contiguity, the typical documentation syntax is abstracted, looking to lower visual impact and reduce meaningless text. Tags that are mandatory by convention, like parameters and return, are automatically generated for non-documented fragments, aside from this, optional tags that can be freely added via a context menu. Tags are fully editable, possessing a text field for the description, as well as for the name, when applicable.

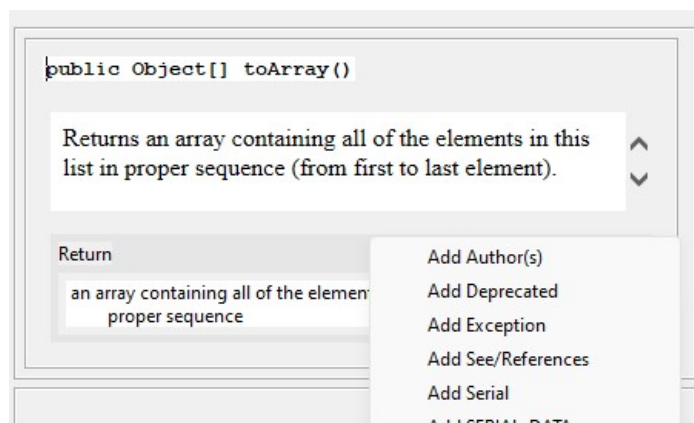


Figure 4.8: The expanded documentation view for the method `toArray`, the context menu enables the expansion of the documentation without using the Javadoc syntax.

Alternatively, to the previously mentioned navigation style, the user can also choose to open a pop-up view from a methods code view to display its documentation. This enables easy documentation editing while coding, without the need to open a dedicated documentation view.

4.3. Implementation

Since PescaJ works as a projectional editor built adhering to the MVC pattern, the model concepts have to be defined (AST). Given that PescaJ is a tool for Java code, we have used JavaParser [3], a well-established library with nearly complete parsing functionality that defines the meta-classes for ASTs of Java files. Hence, our model is a set of JavaParser ASTs. These accept visitors to traverse the parsed code and observers to listen to changes, features that were extensively used in PescaJ. The visitors were used to traverse the ASTs and form the views, whereas the observers enable the views to react to changes in the model (performed in other views).

The code fragment editing widgets of PescaJ are those of Javardise [26], a structured editor for Java, designed to lighten the syntax hurdles on users for didactic purposes. The graphical interface was built using the Standard Widget Toolkit (SWT), to ensure compatibility between the Javardise components, which were also developed using SWT.

PescaJ was developed using Kotlin, not only because of developer preference, but also because Kotlin processes good synergy with Java code, which is the case for the main library used in the project, JavaParser. At the time of writing this document, PescaJ is made up of 2694 lines of code.

4.4. Informal Evaluation

Real user testing was out of scope of this dissertation work. However, it is still possible to submit PescaJ to some informal evaluation. Using the source code for the *ArrayList* class from Java, we can extract lines between methods and documentation line proportion and how this affects navigation on Eclipse, comparing with PescaJ, the number of clicks and scroll used for navigation will also be compared between the two.

4.4.1. Evaluation of code navigation

In the example shown in Figure 4.2, three methods are shown, the method *addAll* makes calls to both *rangeCheckForAdd* and *grow*. These methods are not contiguous in the source files, *addAll* starts at line 730, *rangeCheckForAdd* at line 785 and *grow* at line 241.

In Eclipse, because the representation is tied to the source files, the methods cannot be shown on the same page simultaneously, due to the big distance of lines. To navigate between these methods the developer needs to perform a plethora of navigating actions, the user can employ shortcuts (e.g., *ctrl+click* to snap to method declaration), scroll or search features. However, these actions must be performed every time the user wishes to swap which method is on screen, this can make the number of navigations extremely high, depending on the developer's workflow. An alternative would be to use separate editor, Eclipse allows the user to open the same source file multiple times using split views. This enables the user to place these editors side-by-side, aiding in concurrent editing and visualization of the methods. This approach has limitations, in this example, to have all three methods on screen, the user would need to open three split views at once, as seen in Figure 2.14, this can cause a navigation overhead and add clutter to the working environment.

Using PescaJ, the task of displaying these methods concurrently is made easy, the user can open the methods using the explorer at the left of the editor or by interacting with method calls in the code views. These are then automatically placed in the *workspace* without the need of added navigation actions by the user, with the benefit of reduced clutter. This workflow shows a reduced amount of navigation actions and menu interaction when compared to that of Eclipse, as well as a resulting environment with reduced clutter and more explicit code structure representation, as seen in Figure 4.2.

4.4.2. Evaluation of documentation navigation

PescaJ's documentation capabilities are particularly useful in scenarios where documentation is being created or altered after code has been written. Because documentation is aggregated together without the code, PescaJ is capable of displaying large amounts of method documentation concurrently. In Eclipse the user can edit documentation on the source file, however the documentation is scattered, which hinders the ability to create cohesive documentation with a constant style. Alternatively, the user could open a browser window to visualize the completed documentation, however even an integrated browser does not allow the user to directly edit the documentation.

In PescaJ, the user can navigate directly from the code to its documentation, or alternatively, navigate through the documentation of project packages, this navigation is fast and requires low amounts of input. It is hard to compare this feature to existing tools, not only because it is not present in typical code editors, but also because, at the time of writing, we could not find any work that closely resembles this characteristic of PescaJ.

Conclusions and Future Work

In conclusion, PescaJ serves as a proof of concept for a Java projectional editor, focused on the aggregation of scattered code and documentation fragments. Collocating code and documentation is convenient for associating pieces of API documentation with the respective implementation elements. However, documentation segments stand in the way of code, taking up a considerable amount of lines of code, and consequently contributing to longer scrolls over code files. We measured the number of lines that correspond to *Javadoc* comments in the *java.util* package, and found that, on average, those represent almost half of the file lines. More commercial projects also showed a considerable number of documentation, taking up, on average, around a quarter of the lines of source files. As such, we believe that a form of isolating the two technical concerns (code and documentation) as we propose in PescaJ may contribute to more usable development environments. The documentation views prevent clutter caused by code without the need for additional actions (collapse/uncollapse), as the user can easily explore and edit the aggregated documentation that is scattered in the code base. This feature is especially useful for package and class documentation, whose synthesis is only available in the generated HTML (non-editable).

The study, conducted in Chapter 3, shed a light on common code structure, showing the vast majority of public methods call up to three methods (inside the same project), and have a maximum call depth of four methods. We also observed the majority of method interactions occurred in an intra-class scenario, as opposed to inter-class. These patterns favor the design of PescaJ, given that the developed interface has the capabilities of seamlessly displaying these method structures.

As a proof of concept, PescaJ shows the viability of projectional editors to realize similar features to those of CodeBubbles [2] and PatchWorks [12]. However, PescaJ provides structured editing, automatic fragment placement and specialized views for documentation. This new paradigm of editors could potentially lower user cognitive load, caused by scattered code fragments that can be aggregated by the tools. The kind of juxtaposition provided by tools like PescaJ can hypothetically be beneficial in lowering time spent navigating and improving the process of code understanding. We hope that this work proves to be valuable as one of the stepping stones to the understanding of the usefulness of projectional editors in code development.

5.1. Future Work

The development of PescaJ, as a proof of concept, focused on the innovative features that set it apart from other editors, because of this, some useful features usually present in editors are, at the time of writing this document, not yet implemented. Some other increments are more specific to projectional editors and the unique structure of PescaJ. This section will dive into the future of PescaJ, exploring possible increments, new features and quality of life changes to the editor.

5.1.1. Save State

A feature that is present in almost every work environment is the possibility of saving the current state of the tool, enabling the tool to rebuild the state that was present in a previous work session. This is present in almost every code editor, even the most bare-bones like Notepad++³. Code editors, when reopened, welcome the user with the project loaded and the files that were open last time the editor was running. This feature might be overlooked and forgotten, however it is very noticeable when it is missing, requiring the user to reopen projects and files every time they restart the program.

This feature would be even more useful in a tool like PescaJ. Currently, every time the tool is started, the user is required to browse files to load the project, it becomes even more burdensome when it comes to the *workspaces*, seeing as every method and class needs to be reopened. By saving the state of the tool, PescaJ could easily query the project that was loaded in the last work session, loading it on startup, and *workspaces* could also be rebuilt based on the save-state.

The save state functionality would also allow shareability of workspaces, this could prove very useful for a plethora of use cases, for instance, it would be possible to create a *workspace* demonstrating a bug, with all the relevant methods and classes involved, and then share it across a development team. On the other hand, this could also be used for documentation, creating a *workspace* with the methods used in a feature, making understanding the code easier in the future.

5.1.2. In-editor run-time

The possibility of running the files in the editor is a required feature in IDE's (Eclipse, IntelliJ, etc..). PescaJ would greatly benefit from this feature, drastically reducing the need for other tools in the workflow. This would also enable the creation of run-time views, like variable and other debug information. These views could be integrated into the code views, benefiting from the principle of visual continuity.

³ <https://notepad-plus-plus.org/>

5.1.3. New Views

Due to the nature of projectional editors, it is possible to create new views to display and edit information in alternative ways. Up to this date, the development of PescaJ focused on the organization and interaction between views, only implementing basic views for code and documentation editing. However, for the future of the editor we envision new types of views such as gathering all the implementations of a given interface, or a view addressing member visibility and extensibility (API design). Given that the composition of our views offers some freedom, we also plan to develop ways of forming views from contexts, such as the method calls from an active call stack (debugging session).

Currently the descriptions in all our documentation views are presented and edited as raw text. As future work, these views could be improved if they would use styled text widgets, which would project the documentation text in a more appealing way, avoiding aspects such as “`{@code ...}`” (visible in Figures 4.6, 4.7 and 4.8) and allowing features such as hyperlinks in the text to navigate to related elements. Furthermore, the integration of images in the documentation editors could also be of interest.

5.1.4. Enabling user customization

In projectional editors, views can be freely programmed, feeding off the model and only interacting with a controller. This property makes it easy to use external code to specify views, only needing compliance with the API. With user defined code and a configuration file, PescaJ could easily allow users to code their own views, creating a huge potential for user customization. The downside of this is that the user needs to be code literate to develop their own views, however PescaJ is a tool creator for developers, so the average user would have no problem enhancing their experience in the editor.

5.1.5. Code Elision

Elision is already in use in the documentation views, where only the description of methods is visible, with every field available in an expanded view. The use of elision would also be well employed in the code views, where the full length of code is displayed, even for abnormally long lines or excessively long methods. The implementation of horizontal and/or vertical elision would enable more code fragments to be displayed on screen concurrently, with minimal loss of information displayed.

5.1.6. User Testing

As previously mentioned, real user testing revealed to be out of scope for this project, however, PescaJ would greatly benefit from this approach. User testing would, most likely, reveal some shortcomings of the tool that were not detected in unit testing, like some less intuitive features or bugs. This sort of testing could lead to the improvement of fundamental features in the tool, and as such, should be the next step in the future of PescaJ, before any more features are added. User feedback would also shine a light on the everyday wishes and needs of developers, which could prove useful in the specification of new views.

References

- [1] A. Bragdon et al., “Code bubbles: A working set-based interface for code understanding and maintenance,” Apr. 2010, vol. 4, pp. 2503-2512. doi: 10.1145/1753326.1753706.
- [2] A. Bragdon et al., “Code bubbles: rethinking the user interface paradigm of integrated development environments,” in 2010 ACM/IEEE 32nd International Conference on Software Engineering, 2010, vol. 1, pp. 455–464. doi: 10.1145/1806799.1806866.
- [3] Danny van Bruggen, «javaparser/javaparser: Release javaparser-parent-3.16.1». Zenodo, Mai. 25, 2020. doi: 10.5281/zenodo.3842713.
- [4] R. J. Chevance and T. Heidet, “Static Profile and Dynamic Behavior of COBOL Programs,” SIGPLAN Not., vol. 13, no. 4, pp. 44–57, Apr. 1978, doi: 10.1145/953411.953414.
- [5] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer, “Managerial use of metrics for object-oriented software: an exploratory analysis,” IEEE Transactions on Software Engineering, vol. 24, no. 8, pp. 629–639, 1998, doi: 10.1109/32.707698.
- [6] P. M. Fitts, “The information capacity of the human motor system in controlling the amplitude of movement.,” Journal of Experimental Psychology, vol. 47, pp. 381–391, 1954, doi: 10.1037/h0055392.
- [7] P. M. Fitts and J. R. Peterson, “Information capacity of discrete motor responses.,” Journal of Experimental Psychology, vol. 67, pp. 103–112, 1964, doi: 10.1037/h0045689.
- [8] M. Fowler, “ProjectionalEditing”, martinowler.com, <https://martinowler.com/bliki/ProjectionalEditing.html#:~:text=With%20projectional%20editing%20the%20abstract,the%20definition%20of%20the%20system> (accessed Jan. 25, 2023).
- [9] J. Y. Gil and I. Maman, “Micro patterns in Java code,” in OOPSLA ’05: Proceedings of 20th ACM SIGPLAN conference on Object oriented programming systems languages and applications. ACM Press, 2005, pp. 97–116. doi: 10.1145/1167515.1167507.
- [10] R. Harrison, S. Counsell, and R. Nithi, “Coupling metrics for object-oriented design,” in Proceedings Fifth International Software Metrics Symposium. Metrics (Cat. No.98TB100262), 1998, pp. 150–157. doi: 10.1109/METRIC.1998.731240.
- [11] A. Z. Henley and S. D. Fleming, “The Patchworks Code Editor: Toward Faster Navigation with Less Code Arranging and Fewer Navigation Mistakes,” in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2014, pp. 2511-2520. doi: 10.1145/2556288.2557073.
- [12] A. Z. Henley, S. D. Fleming, and M. V. Luong, “Toward Principles for the Design of Navigation Affordances in Code Editors: An Empirical Investigation,” in Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, 2017, pp. 5690-5702. doi: 10.1145/3025453.3025645
- [13] D. E. Knuth, “An empirical study of FORTRAN programs,” Software–Practice and Experience, vol. 1, no. 2, pp. 105–133, 1971. doi: 10.1002/spe.4380010203

- [14] A. J. Ko, H. H. Aung, and B. A. Myers, "Design Requirements for More Flexible Structured Editors from a Study of Programmers' Text Editing," in CHI '05 Extended Abstracts on Human Factors in Computing Systems, 2005, pp. 1557-1560. doi: 10.1145/1056808.1056965.
- [15] A. Ko and B. Myers, "Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors," in Conference on Human Factors in Computing Systems - Proceedings, Apr. 2006, vol. 1, pp. 387-396. doi: 10.1145/1124772.1124831.
- [16] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks," IEEE Transactions on Software Engineering, vol. 32, no. 12, pp. 971-987, 2006, doi: 10.1109/TSE.2006.116.
- [17] Krasner, G. E., & Pope, S. T. (1988). A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80. J. Object Oriented Program., 1(3), 26-49, doi: 10.5555/50757.50759.
- [18] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of Application Software Maintenance," Commun. ACM, vol. 21, no. 6, pp. 466-471, Jun. 1978, doi: 10.1145/359511.359522.
- [19] J. F. Lopes and A. L. Santos, "Pescal: A Projectional Editor for Java Featuring Scattered Code Aggregation", Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT), SPLASH'23 Workshops, Oct. 2023, doi: 10.1145/3623504.3623571.
- [20] I. S. MacKenzie and W. Buxton, "Extending Fitts' Law to Two-Dimensional Tasks," in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 1992, pp. 219-226. doi: 10.1145/142750.142794.
- [21] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," Commun. ACM, vol. 33, no. 12, pp. 32-44, Dec. 1990, doi: 10.1145/96267.96279.
- [22] R. Minelli, A. Mocci, and M. Lanza, "I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time," in 2015 IEEE 23rd International Conference on Program Comprehension, 2015, pp. 25-35. doi: 10.1109/ICPC.2015.12.
- [23] A. Nemes, "4 Tips to Combine Text and Graphics for Better eLearning", elearningindustry.com, <https://elearningindustry.com/4-tips-combine-text-and-graphics-for-better-elearning> (accessed Jan. 29, 2023).
- [24] K. Peffers et al., "Design Science Research Process: A Model for Producing and Presenting Information Systems Research," 2020, doi: 10.48550/ARXIV.2006.02763.
- [25] M. Plumlee and C. Ware, "Zooming versus multiple window interfaces: Cognitive costs of visual comparisons," ACM Trans. Comput.-Hum. Interact., vol. 13, pp. 179-209, Jan. 2006.
- [26] A. L. Santos, "Javardise: A Structured Code Editor for Programming Pedagogy in Java," in Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming, 2020, pp. 120-125. doi: 10.1145/3397537.339756.

- [27] L. Schroeder and A. T. Cenkci, "Spatial Contiguity and Spatial Split-Attention Effects in Multimedia Learning Environments: a Meta-Analysis," *Educational Psychology Review*, vol. 30, no. 3, pp. 679–701, Sep. 2018, doi: 10.1007/s10648-018-9435-9.
- [28] E. Tempero et al., "The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies," in *2010 Asia Pacific Software Engineering Conference*, 2010, pp. 336–345. doi: 10.1109/APSEC.2010.46.
- [29] M. Voelter, J. Siegmund, T. Berger, and B. Kolb, "Towards User-Friendly Projectional Editors," in *Software Language Engineering*, 2014, pp. 41–61. doi: 10.1007/978-3-319-11245-9_3.
- [30] R. Wheeldon and S. Counsell, "Power law distributions in class relationships," *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 45–54, 2003, [Online]. Available: <https://api.semanticscholar.org/CorpusID:8795254>.
- [31] S. W. L. Yip and T. Lam, "A software maintenance survey," in *Proceedings of 1st Asia-Pacific Software Engineering Conference*, 1994, pp. 70–79. doi: 10.1109/APSEC.1994.465272.