# PescaJ: A Projectional Editor for Java Featuring Scattered Code Aggregation

José F. Lopes
josefaulopes@gmail.com
Instituto Universitário de Lisboa (ISCTE-IUL)
Lisboa, Portugal

André L. Santos
andre.santos@iscte-iul.pt
Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR-IUL
Lisboa, Portugal

## Abstract

Conventionally, source code (and its documentation) 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 folds on imports, documentation, methods, etc). This is a sign that the developers seek more convenient code editing forms than the direct manipulation of text files.

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. Furthermore, we provide documentation editors, also aggregating scattered Javadoc comments, that can be used in parallel with source code editing.

***CCS Concepts:*** • **Human-centered computing → Interactive systems and tools**; • **Software and its engineering → Software maintenance tools**.

***Keywords:*** Projectional editors, separation of concerns, documentation, Java

**ACM Reference Format:**
José F. Lopes and André L. Santos. 2023. PescaJ: A Projectional Editor for Java Featuring Scattered Code Aggregation. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT '23), October 23, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3623504.3623571


*PAINT '23, October 23, 2023, Cascais, Portugal*

ACM ISBN 979-8-4007-0399-7/23/10…$15.00
https://doi.org/10.1145/3623504.3623571

## 1 Introduction

Conventionally, source code files are simultaneously a storage and editing representation, the latter performed by text editors (albeit typically with sophisticated support). The nature of text files imposes a linear representation dictated by the sequence of lines, which in turn directly relates to the memory chunk that is used to store that information (sequence of bytes). However, the dependencies between the members of a class most likely do not align with the order in which they are in the file. For example, a class containing three methods whose dependencies form a chain $m1 \rightarrow m2 \rightarrow m3$ could match the order in the file, but these cases are not the norm.

Developers spend much of their time exploring and understanding the existing code base [9]. Frequently, a method may depend on another one many lines apart, leading to scroll activity when inspecting them in sequence. Studies have shown that up to a third of developers' time in IDEs is spent on code navigation activities [5]. Projectional editors [13] are a technique that can break up this linearity, where the editing representation is distinct from the storage representation. In this way, code *views* may be formed in more convenient ways that reduce navigation.

*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.

In our approach, we address the separation of *technical* concerns. We consider the implementation of reusable code and its documentation to be two distinct technical concerns. There is in fact a developer role focusing on documentation — the *technical writers* — which may even be a part of independent teams from those of implementation [3]. For API documentation purposes, code artifacts conventionally hold both implementation and documentation in source code files — using tools such as Javadoc and Doxygen) — despite the existence of other external documentation artifacts available

(e.g., wikis, diagrams). The two activities of implementing and documenting may not occur at the same stage, most notably the documentation that targets the API of reusable software (library or framework). A study of how the documentation of Eclipse evolves confirmed this fact [11].

We present PescaJ, a prototype projectional editor for Java where a developer is able to quickly form different types of editable views that aggregate parts of code or documentation, which otherwise in conventional settings are scattered across the code files. Different views may overlap with respect to the elements they present, which are kept in sync through a Model-View-Controller architecture (see Figure 1). For example, one may be editing methods of a class in a view, while having another view to edit the whole documentation of that same class. In this paper, we describe the general architectural approach and its materialization in the PescaJ views and their rationale.

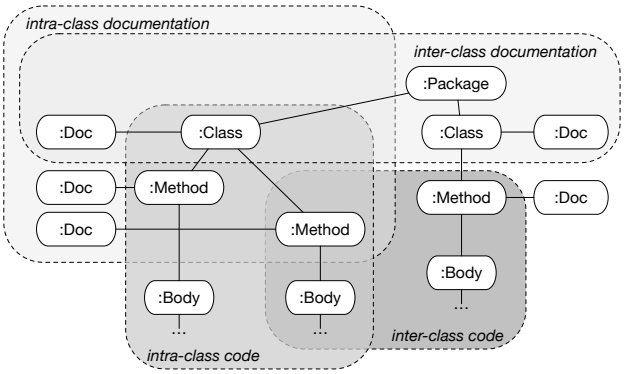


**Figure 1.** PescaJ enables overlapping views over shared elements of ASTs.

## 2 Related Work

PescaJ's goals are closely related to those of CodeBubbles [2], but the latter is not built as a projectional editor. Code fragments, called bubbles, can be opened in a virtual environment, where dependencies between methods are shown by an arrow that links the method's respective bubble. Grouping and placement of bubbles can be formed and rearranged freely by the user. This gives developers more freedom over their workspace, however, it may introduce some overhead in the time used for navigating, since some organization of the code fragments is done by the user, and not by the tool. With these features, CodeBubbles may be effective at juxtaposing relevant code fragments, at the expense of some reorganizing by the user. However, some elements of the code are still somewhat difficult to visualize, seeing as there is no easy way to aggregate them. For instance, it is burdensome for a developer to write or consult the documentation of the classes.

PatchWorks [4] takes a more structured approach, code fragments are shown in patches, which are arranged in a two-tall virtual ribbon. There are two main ways to visualize the ribbon, a zoomed-in grid view, where 6 patches can be seen concurrently, or a zoomed-out view of the ribbon, where the user can select the target for the zoomed view. However, much like CodeBubbles, PatchWork's workspace organization is placed on the user, lacking an automatic way to juxtapose related information. PatchWorks also does not work as a projectional editor, staying confined to the typical text representation of code fragments.

Structured editors [8] restrict the code editing manipulations so that syntactical correctness is maintained. Barista [6] is a structured editor where certain code elements can be represented in alternative views, such as integrated images for documentation or tables for boolean expressions. In our work, the focus is not on enriching views with additional elements or alternative ones, but rather on views that gather information and allow editing information that is otherwise scattered throughout several implementation files. Sandblocks [1] is a tool that is able to generate structured editors from grammar descriptions, avoiding the need to "hand-craft" structured editors.

Projectional editors [13] are a technique to implement structured editors, possibly displaying code in different forms other than code. Jetbrains' Meta Programming System (MPS)[1] is an industrial tool that allows such views to be defined, focusing on the definition of Domain-Specific Languages (DSLs). In MPS, Abstract Syntax Trees (AST) are represented in a tool-specific format that is used by DSL engineers to define the concepts of a DSL, and in turn, define their editors (textual or graphical). The views we propose are not available in MPS, but in principle, they could be implemented using the platform to define alternative editors for Java.

Popular IDEs (e.g. Eclipse[2], IntelliJ[3], VSCode[4]) feature *split views*, where one file may be edited simultaneously in more than one view, keeping them in sync. This feature may be used to achieve a similar form of juxtaposing methods to what we propose in PescaJ. However, using it is not very practical as it requires a bit of setup to achieve views like for instance the one we illustrate in Figure 4. Moreover, IDEs do not feature the notion of a workspace, which in our approach may evolve by easily adding and removing elements as needed.

Most IDEs also feature code folding facilities to crop segments to reduce the "height" of a file. This helps with respect to the amount of scrolling, but it does not avoid that large navigation jumps have to be performed to go through the members of a class. Good documentation should be consistent with respect to terminology and style. Having pieces

[1]https://www.jetbrains.com/mps
[2]https://www.eclipse.org
[3]https://www.jetbrains.com/idea
[4]https://code.visualstudio.com

of text spread throughout regions of a file or across several files is not the most practical form of achieving this. To our knowledge, there are no available tools that allow developers to edit the embedded documentation using a separate editing abstraction that does not require manipulating the source code directly. We are confident that such a tool would not be technically difficult to achieve through a custom editor that performs AST rewriting (offline). However, we argue that a solution to keep code and documentation views seamlessly synchronized should be more challenging to accomplish with conventional source code editors.

The development of PescaJ took into account some Human-Computer interaction principles. Spatial Contiguity or Spatial Split Attention Effect [12] are concepts that explain that, while learning, when a subject is required to split their attention between pieces of information, they suffer higher cognitive load and impaired learning, when compared to scenarios where information is properly juxtaposed. These principles can be applied to code editors, where developers may need to switch files or use the scroll while analyzing previously written code. Fitt's law is a concept of ergonomics, that states that the smaller and farther a target is from the pointer, the more likely the user is to make a pointing error [7]. Accurate pointing is a key part of fast navigation, meaning that code fragments that developers may click consecutively should be sized and placed taking Fitt's law into account.

## 3 Approach

The main goal of our approach is to reduce the user's cognitive load and time spent navigating while coding, by visually aggregating closely related and relevant information. The code is displayed to the user in fragments, grouped in views that aggregate scattered parts of the source code. We seek not only to lower the split attention effect but also to increase the user's accuracy while navigating with the mouse, according to Fitt's law.

Figure 2 presents an illustrative package, holding three classes with documentation (grey parts), to introduce PescaJ's views. So far, we implemented different views that can be split into two categories — *code* and *documentation* — which in turn, may be *intra* or *inter-class*. Figure 3 schematically presents which fragments are presented and edited by our views, referring to the same elements of Figure 2. Notice that the same element may appear in different views under different perspectives.

We propose to realize these sorts of views using an architecture based on Model-View-Controller (MVC), aiming at having multiple simultaneous views with overlapping model elements. The overall *model* is the set of ASTs of the code files, whereas the views address fragments of one or more ASTs, not the trees as a whole — a key characteristic of our approach. Following the MVC principles, each view observes

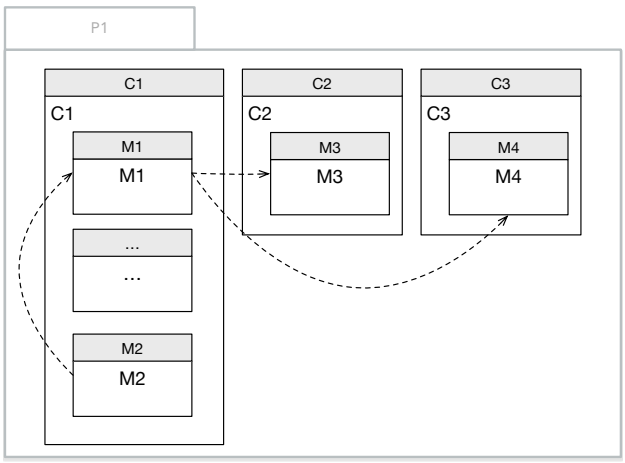


**Figure 2.** Example package (P) containing classes (C) with methods (M). Documentation segments are depicted in grey, dashed arrows represent dependencies.

the nodes under interest and reacts to changes when notified, whereas editing actions are represented in *commands* whose execution is centralized in a controller. Hence, when a command modifies an AST node, all the views that registered observers in that node will react to those changes.

Figure 1 illustrates how different views refer to shared model elements (AST nodes). As an example, a view of class documentation may be displaying a list of its method names and their documentation, while observing changes in the list of members of the AST and their corresponding documentation node, in order to update the view content accordingly. In turn, a code editing view may be editing that same class, and its commands will modify the class AST. For example, the documentation view may modify a method's documentation without any impact on the code view, because the latter is neither displaying nor observing documentation AST nodes. Analogously, if the code editing view modifies a method body, the documentation will remain as is. However, if the code view renames a method or adds a new one, the documentation will react to those changes because both views are referring to the same AST nodes (list of class members, method, and documentation nodes).

## 4 PescaJ

Figure 4 presents a screenshot of PescaJ. In the left-hand part, we can see a conventional package explorer view similar to those available in most IDEs. PescaJ employs workspaces, which are similar to the concept of *working set* of CodeBubbles [2]. These are canvases that hold different views, which are populated either by dragging elements from the package explorer or expanding method calls within the view. They enable users to create clusters of information readily available, for example, a developer may use a workspace to edit the code of a class and another to visualize and edit its

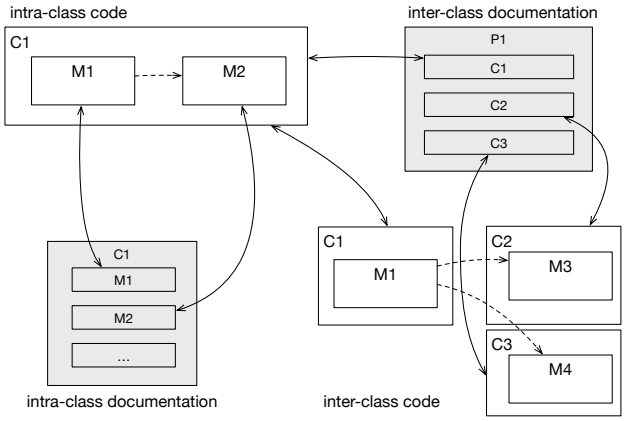


**Figure 3.** Views for aggregating related code and untangling code and documentation using the elements of Figure 2. Bidirectional arrows represent elements that are kept in sync.

documentation. Alternatively, a developer could use different workspaces for working on distinct features in parallel without scrolling within or switching between files.

PescaJ works as a projectional editor, but its views do not resemble conventional text editors where a whole class source is being edited. Instead, the views display *fragments* of classes (i.e., only a part of their members), while they may aggregate parts of different classes. So far, we developed two main sorts of views: *code* and *documentation*.

### 4.1 Code Views

The code views allow the user to visualize and edit the code, either *intra-class*, meaning that they display fragments of a class, or *inter-class*, meaning that they display fragments of several classes. 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 by interacting with the current code. If the user wants to visualize a called method, the editor widget for editing that method's code will be placed near the call site (at its right), following the principle of visual contiguity. In this way, a chain of dependent methods is visualized from left to right in the workspace.

Figure 4 shows three methods of the class *ArrayList* (Java libraries), annotated with the line distances in the source code file. The method *addAll* calls the method *rangeCheckForAdd*, clicking the call opens a view to the right containing the latter. The method *grow* is called as well, which can also be opened to continue expanding the call graph. The method *grow* is called a few lines after *rangeCheckForAdd*, and so the corresponding view is placed below the latter, maintaining a consistent order.

Juxtaposing methods is not limited to fragments within the same class, it can also represent dependencies between inter-class methods, creating multiple class widgets that are placed following the same logic seen in the last example. For example, as illustrated in Figure 5, by clicking the method call inside *toArray* of the class *ArrayList*, 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.

### 4.2 Documentation Views

Documentation views are responsible for aggregating documentation parts that are scattered throughout several regions in the code. While using these views, the user can visualize and edit the documentation of the chosen code fragments, making tasks like comparing documentation style and coherence easier. PescaJ provides four documentation views, which resemble the decomposition of the documents generated by Javadoc: *set of packages*, containing the package descriptions; *package* (see Figure 6), containing a list of classes with their documentation header; *class* (see Figure 7), containing a list of methods and the header of their documentation; *method* (see Figure 8), containing all the detailed documentation of a method.

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 that have public non-documented methods contain a warning, allowing the user to know what classes need to be documented, without browsing each one individually for performing edits. When hovering the warning additional information is presented, such as the number of public non-documented methods as seen in Figure 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).

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 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 7. As an alternative to the documentation views, the user can open a pop-up view from a code view to display and edit documentation. This enables easy documentation editing while coding, without the need to open a dedicated documentation view.

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 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 clutter. Tags that are mandatory by convention, such as those for

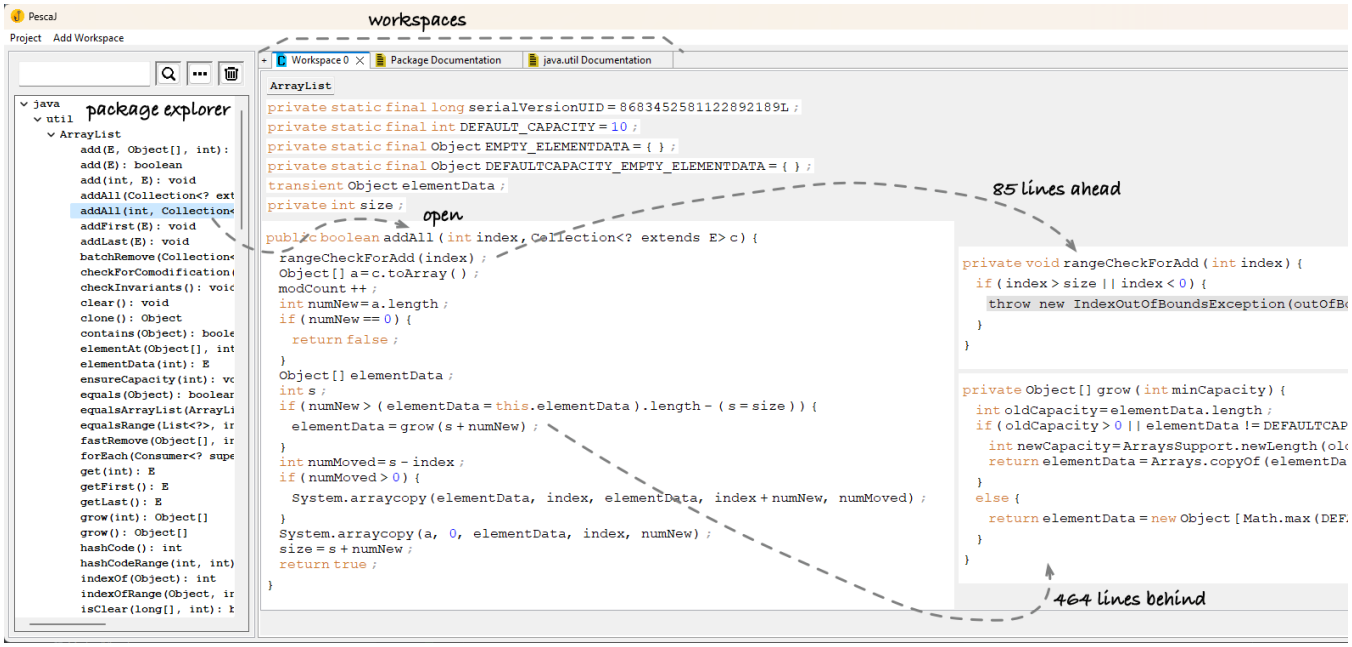


**Figure 4.** PescaJ displaying an intra-class view illustrated with a fragment of the ArrayList class (Java libraries), juxtaposing methods according to call dependencies.

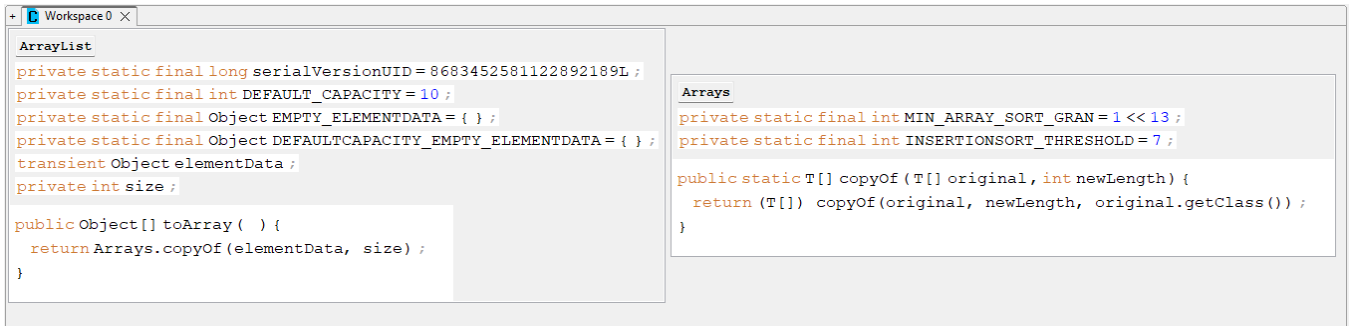

**Figure 5.** An inter-class view illustrated with a fragment of the classes ArrayList and Arrays (Java libraries), where the method toArray depends on method copyOf, which are juxtaposed.

parameters and return, are automatically generated for non-documented fragments. Additionally, optional tags can be added via a context menu. Tags are fully editable, possessing a text field for the description, as well as for the name, when applicable.

### 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 [14][5], 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 enabled the views to react to changes in the model (performed in other views).

The code fragment editing widgets of PescaJ are those of Javardise [10], 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.

## 5 Discussion

Colocating code and documentation is convenient for associating pieces of API documentation with the respective implementation elements. However, documentation segments

[5]javaparser.org

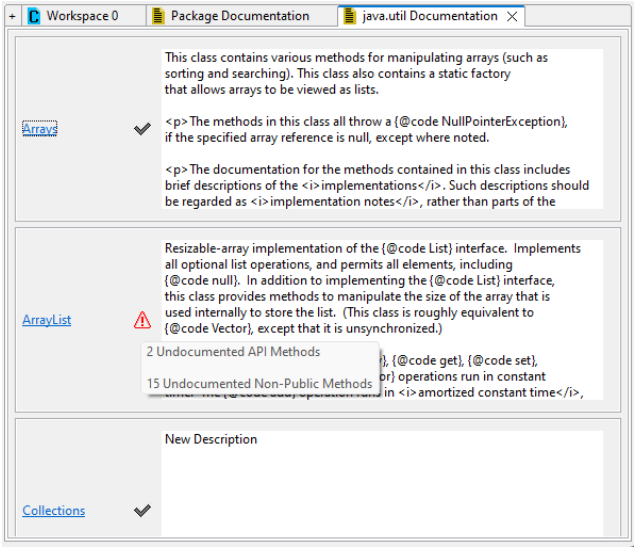


**Figure 6.** View that aggregates the documentation of classes in a package. The warning shows that the class ArrayList contains public undocumented methods.

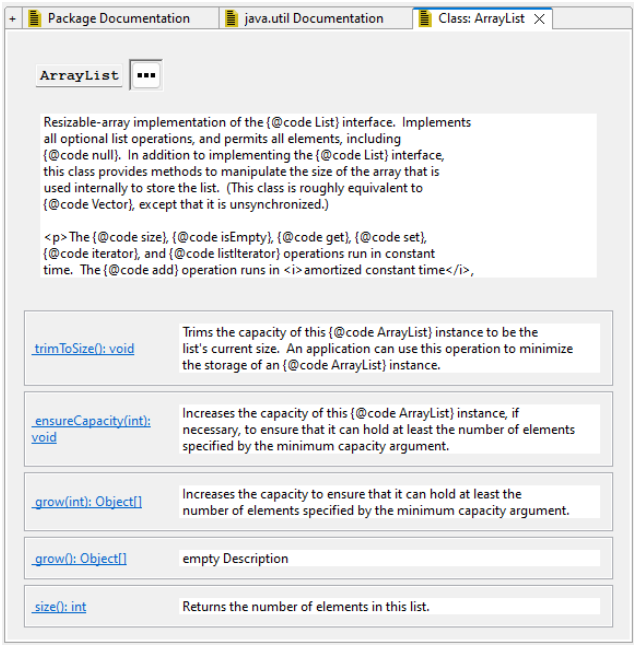


**Figure 7.** View that aggregates the documentation of the methods in a class, showing a documentation header for each method.

stand in the way of code, taking up a considerable amount of lines of code, and consequently contributing to longer scrolls over code files. As an example, we measured the number of lines that correspond to Javadoc comments in the `java.util` package (source code of OpenJDK 12+32[6]), and found that

[6]https://github.com/openjdk/

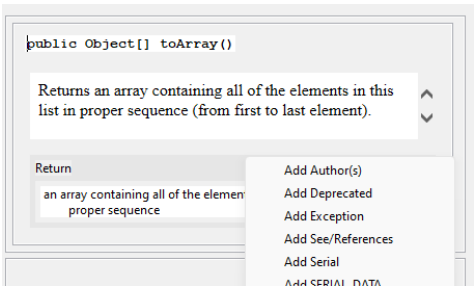


**Figure 8.** The expanded documentation view for the method toArray, the context menu enables the expansion of the documentation without using the Javadoc syntax.

on average those represent almost half of the file lines (approximately 46%, excluding empty lines). 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).

As a proof of concept, PescaJ shows the viability of projectional editors to realize similar features to those of CodeBubbles [2] and PatchWorks [4]. However, PescaJ provides structured editing 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. In future work, we plan to carry out user experiments to assess the efficiency of software development tasks using the proposed views.

Regarding increments to PescaJ, 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).

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 the figures) 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.

## Acknowledgments

We are grateful to the anonymous reviewers for their valuable suggestions to improve this paper. This work was partially supported by Fundação para a Ciência e a Tecnologia, I.P. (FCT) [ISTAR Projects: UIDB/04466/2020 and UIDP/04466/2020].

## References

[1] Tom Beckmann, Patrick Rein, Stefan Ramson, Joana Bergsiek, and Robert Hirschfeld. 2023. Structured Editing for All: Deriving Usable Structured Editors from Grammars. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 595, 16 pages. https://doi.org/10.1145/3544548.3580785

[2] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. 2010. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. 455–464. https://doi.org/10.1145/1806799.1806866

[3] Barthélémy Dagenais and Martin P. Robillard. 2010. Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. Association for Computing Machinery, New York, NY, USA, 127–136. https://doi.org/10.1145/1882291.1882312

[4] Austin Z. Henley and Scott D. Fleming. 2014. 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 (CHI '14)*. Association for Computing Machinery, New York, NY, USA, 2511–2520. https://doi.org/10.1145/2556288.2557073

[5] A.J. Ko, Htet Htet Aung, and B.A. Myers. 2005. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 126–135. https://doi.org/10.1109/ICSE.2005.1553555

[6] Amy J. Ko and Brad A. Myers. 2006. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '06)*. Association for Computing Machinery, New York, NY, USA, 387–396. https://doi.org/10.1145/1124772.1124831

[7] I. Scott MacKenzie and William Buxton. 1992. Extending Fitts' Law to Two-Dimensional Tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '92)*. Association for Computing Machinery, New York, NY, USA, 219–226. https://doi.org/10.1145/142750.142794

[8] Philip Miller, John Pane, Glenn Meter, and Scott Vorthmann. 1994. Evolution of Novice Programming Environments: The Structure Editors. In *of Carnegie Mellon University*. 140–158.

[9] Roberto Minelli, Andrea Mocci and, and Michele Lanza. 2015. I Know What You Did Last Summer: An Investigation of How Developers Spend Their Time. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension (ICPC '15)*. IEEE Press, 25–35.

[10] André L. Santos. 2020. 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 (Programming '20)*. Association for Computing Machinery, New York, NY, USA, 120–125. https://doi.org/10.1145/3397537.3397561

[11] Daniel Schreck, Valentin Dallmeier, and Thomas Zimmermann. 2007. How Documentation Evolves over Time. In *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting (IWPSE '07)*. Association for Computing Machinery, New York, NY, USA, 4–10. https://doi.org/10.1145/1294948.1294952

[12] Noah L. Schroeder and Ada T. Cenkci. 2018. Spatial Contiguity and Spatial Split-Attention Effects in Multimedia Learning Environments: a Meta-Analysis. *Educational Psychology Review* 30, 3 (2018), 679–701. https://doi.org/10.1007/s10648-018-9435-9

[13] Charles Simonyi, Magnus Christerson, and Shane Clifford. 2006. Intentional Software. *SIGPLAN Not.* 41, 10 (oct 2006), 451–464. https://doi.org/10.1145/1167515.1167511

[14] Danny van Bruggen, Federico Tomassetti, Roger Howell, Malte Langkabel, Nicholas Smith, Artur Bosch, Malte Skoruppa, Cruz Maximilien, ThLeu, Panayiotis, Sebastian Kirsch (@skirsch79), Simon, Johann Beleites, Wim Tibackx, jean pierre L, André Rouél, edefazio, Daan Schipper, Mathiponds, Why you want to know, Ryan Beckett, ptitjes, kotari4u, Marvin Wyrich, Ricardo Morais, Maarten Coene, bresai, Implex1v, and Bernhard Haumacher. 2020. javaparser/javaparser: Release javaparser- parent-3.16.1. https://doi.org/10.5281/zenodo.3842713

Received 2023-07-17; accepted 2023-08-07