Bridging Incremental Programming and Complex Software Development Environments

Max Boksem
max.boksem@student.uva.nl
University of Amsterdam
Amsterdam, Netherlands

L. Thomas van Binsbergen
ltvanbinsbergen@acm.org
University of Amsterdam
Amsterdam, Netherlands

Abstract

In modern software development, programmers typically choose between two main types of coding environments: Incremental Programming Environments (IPEs), such as the Read-Eval-Print-Loop (REPL) interpreter IPython and the Jupyter Computational Notebook, and Integrated (text-based) Development Environments (IDEs), such as Visual Studio Code. IPEs excel in providing immediate feedback for iterative development, testing, and debugging, making them ideal for fields like data science and AI. However, their typically linear and isolated interface struggles with managing the complexity of larger software projects. Conversely, traditional IDEs support extensive project management and debugging tools suited for complex applications but lack the interactive and incremental nature of IPEs.

This paper reports on an ongoing investigation and design of a hybrid environment that combines benefits of IPEs and IDEs and the programming styles they naturally support. Central to our design is a graph structure representing code fragments as nodes and code structure as edges. By considering various types of nodes and relationships (e.g. for representing class membership, execution order, documentation, and dependencies) we can facilitate aspects of both incremental programming (in IPEs) and complexity management (in IDEs). We demonstrate our approach with a prototype, called Incremental Graph Code (IGC), by presenting its architecture and a showcase. We demonstrate IGC's functionality and discuss its potential advantages over existing environments. Key features include advanced code visualization, modular and incremental development, and complexity management. IGC aims to provide a unified, extensible, and flexible development environment that bridges the gap between different styles of programming.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PAINT '24, October 22, 2024, Pasadena, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1212-8/24/10
https://doi.org/10.1145/3689488.3689991

CCS Concepts: • **Software and its engineering → Integrated and visual development environments**; **Software development techniques**.

Keywords: incremental programming, exploratory programming, read-eval-print loop, visual general-purpose language development environment, software complexity management

ACM Reference Format:
Max Boksem and L. Thomas van Binsbergen. 2024. Bridging Incremental Programming and Complex Software Development Environments. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT '24), October 22, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3689488.3689991

1 Introduction

There are many software development environments for programmers to choose from. Depending on the user's use case, they might want to choose an Incremental Programming Environment (IPE) geared towards immediate feedback and experimentation [27], or a robust Integrated Development Environment (IDE) geared for full-blown software engineering projects of all scales. Each type of programming environment offers distinct advantages but also reveals limitations when applied outside their ideal contexts.

IPEs, such as the Read-Eval-Print-Loop (REPL) interpreter IPython and the Jupyter Computational Notebook, are excellent in providing immediate feedback, which is especially helpful for iterative development, initial experimentation, testing, and debugging. Computational Notebooks such as those provided by Jupyter [16], Matlab[1], and R-studio, also admit forms of exploratory programming by allowing code cells to be re-executed and re-ordered. Exploratory programming is a more general form of incremental programming in which the end goal is not known upfront and instead discovered through the interaction with code [1, 23, 26]. Experimentation and exploration have been found to be beneficial to data science and computational science [13, 15]. However, Notebooks are also found to be limited with regards to exploratory programming [7, 12, 14]. Moreover, the linear and isolated interface of REPLs and Notebooks can become a liability for larger projects. Code fragments are typically

[1] https://nl.mathworks.com/products/matlab/live-editor.html

provided in isolated cells, without ways of structuring or grouping fragments together (e.g., to form modules, classes or packages), which can lead to difficulties in maintaining the complexity of traditional software projects. This property hinders the application of IPEs to complex software systems where managing relationships and dependencies between code fragments, i.e., the structure of the source code itself, are essential.

Traditional text-based environments and IDEs, such as Visual Studio Code (VSCode), are seen as the standard of large-scale software development (as evidenced by Stackoverflow developer survey of 2024[2]). These environments usually support extensive project management and debugging tools designed for complex applications. Nevertheless, they lack the interactive and incremental nature of IPEs, which are especially beneficial in the initial phases of development.

Contribution. We demonstrate a new hybrid environment, in the form of a prototype named Incremental Graph Code (IGC)[3], reflecting the current state in an on-going investigation into novel programming environments that combine benefits of both IPEs and IDEs. The goal of the investigation is to explore the extent to which a unified programming experience – suitable for many types of developers, development tasks, and development stages – can be realized within a single environment. Our investigation is centered around a novel way to represent source code, inspired by PescaJ, presented at PAINT last year [19], in which code fragments are organized as nodes in a graph. The graph consists of several types of nodes, e.g. with a node type for documentation in addition to a node type for code. Similarly, different edge types are supported to represent different relationships between nodes (and the code fragments they hold). For example, one edge type links documentation to a code fragment, a second links a method to a class, and a third represents execution order (starting from a special initial state node). The screenshot in Figure 1 shows a visualization of the graph used by IGC and demonstrates some of the mentioned node and edge types.

Our method of investigation involves the iterative extension of our IGC prototype with additional features comprised of node types and edge types, along with their front end visualisations and back end effects. The strength of the approach is witnessed by the interactions that different features afford, e.g. exploratory programming in a notebook can be realized using the combination of code-nodes, documentation-nodes, output visualization-nodes, and execution edges. In this paper we showcase recreating a typical Jupyter Notebook environment within IGC. In future work we will further evaluate our approach with additional case studies, e.g., by demonstrating the capabilities of PescaJ for creating documentation-oriented views.

In Section 2, we introduce background concepts that inspired the main features and requirements that motivated specific design decisions. In Section 3 and 4 we present the general architecture and design or implementation decisions encountered. Section 5 performs the showcase, highlighting many of the features discussed previously and demonstrating the advantages IGC can offer.

2 Background

The following section introduces background knowledge and describes related work from which we derive key requirements for our design in Section 3.

2.1 REPLs and Computational Notebooks

Incremental programming is an approach to software development where the process is segmented into small, manageable programs that are built on top of one another. This differs from normal development, which focuses on writing a huge code feature and then testing everything afterward. Developing incrementally can help alleviate bugs faster, therefore making development more efficient.

Incremental programming shines in environments such as Read-Eval-Print Loops (REPLs) [20, 27]. Often seen in easily accessible environments, such as in command-line interfaces, REPLs are characterized by the iterative development style they afford, gradually building up a software project piece by piece, allowing for continuous testing, adaptation, and refinement. The interpreters (typically) underlying REPL environments, can be further incorporated for use in computational Notebooks, such as Jupyter Notebooks [16]. Computational notebooks have become increasingly popular, especially in the fields of data science. This is shown by the rapid growth in the availability of Jupyter Notebooks on GitHub, soaring from about 200,000 in 2015 to nearly 10 million[4] by October 2020 [22]. The Jupyter extension for VSCode is also the third most downloaded extension, with more than 74 million downloads at the time of writing[5].

REPLs have a lot of advantages for code development, such as quick feedback when manipulating code fragments, quick testing of code snippets (e.g. calls to library functions), and modularity of code fragments. Notebooks bring additional advantages through their mixture of code and documentation cells, enabling literal programming [17], and through output-cells that can visualize the output of code execution, sometimes even affording interaction (e.g. as slider).

[2]https://survey.stackoverflow.co/2024/
[3]https://github.com/MaxMB15/MSc-SE-Master-Project/tree/main/IncrGraph
[4]https://blog.jetbrains.com/datalore/2020/12/17/we-downloaded-10-000-000-jupyter-notebooks-from-github-this-is-what-we-learned/
[5]https://marketplace.visualstudio.com/vscode

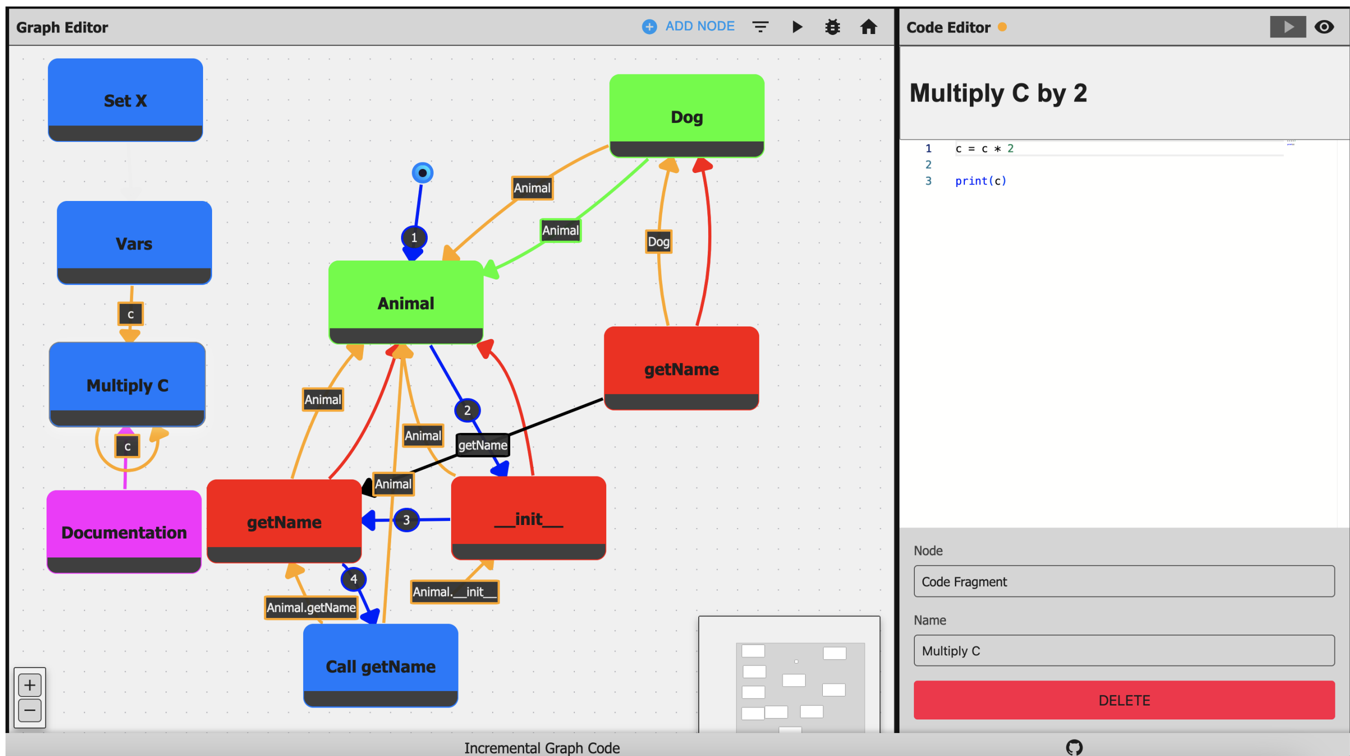

Figure 1. General view of IGC showing the Graph and Code editor. Displayed in this image are four node types: Code Fragment (blue), Documentation (pink), Class (green), and Method (red). There are six main relationships shown: Execution (blue), Dependency (orange), Documentation (pink), Method (red), Inheritance (green), and Overrides (black).

IPEs are usually limited to relatively small applications, such as the pipelines developed by data scientists and programs for educational purposes. A complex project can be implemented entirely in a REPL or Notebook, but this is similar to having all code within a single file, which is impractical to read and analyze. For all the benefits IPEs have to offer, they cannot be utilized effectively in the context of software development outside of smaller software projects.

2.2 Exploratory Programming

The style of exploratory programming involves experimenting with different versions of the same code fragment and responding to the observed effects (e.g. modifying and executing a code cell in a notebook multiple times). Various approaches have been investigate to better support exploratory programming in Notebooks [13] or programming environments in general [26]. One approach to supporting exploratory programming is to allow developers to manipulate execution paths to explore and understand alternative branches of program evolution. This approach enables developers to test different scenarios by altering the sequence of code execution. Instead of modifying and executing one specific cell, a developer might also choose to execute an alternative cell to observe how the two compare based on their outputs. This flexibility facilitates deeper insights into how various parts of the code interact. Ideally, outputs from these different execution paths can be easily compared, providing clear and immediate feedback on how changes impact the code's behavior [23]. In [9], a generic approach for supporting exploratory programming in programming environments is suggested in which an execution graph keeps track of previously visited program states to which the programmer can return, similar to the record-keeping required for omniscient debugging [3].

2.3 Classical Text-based Software Development

Classical text-based development practices have been the norm for creating large-scale software projects. A developer opens an IDE or text editor, writes some code, and clicks run, typically invoking a compiler to build and run an executable (i.e., 'edit-compile-run' instead of 'read-eval-print'). This method is adaptable and offers flexibility to almost every coding application. It does have its pitfalls, however. As a software project's complexity increases, readability tends to decrease [24]. A large area of research is concerned with how to convert classical development methods into more readable forms such as Domain-Specific Languages (DSLs), UML generators, and Model Driven Development (MDD). Integrating incremental programming into this framework could address some of these concerns by offering developers

the advantage of immediate code feedback and simplified experimentation with code modifications.

3 Design

This section outlines the requirements and design goals for our prototype IGC. The requirements are derived from literature and existing programming environments as described in the previous section.

3.1 Requirements

Several high-level requirements must be met to create a hybrid programming environment that leverages the benefits of incremental programming and traditional text-based development. This environment should provide the interactive and iterative features of REPLs, allowing developers to **test and refine their code in small, manageable increments**. Simultaneously, it should offer the **structural support and comprehensive project management features** found in classical text-based environments, making it suitable for large-scale software projects.

The environment should include features that help manage dependencies, track changes, and organize code effectively to **handle the complexity** of large software projects. By **representing code and its relationships in a graph-like format**, we can categorize and organize different sections of a project more effectively. This graph structure enhances readability and manageability as it can be designed to look like popular **system design structures**, such as UML. IGC should support different architectural views to visualize the structure and relationships of the code in various forms.

Each node within the graph should be capable of independent execution, adhering to the principles of **incremental programming and modular development**. This capability ensures that developers can run, edit, test, and analyze nodes in isolation. Additionally, the environment must include version control integration to facilitate collaborative work and ensure that changes are tracked efficiently.

File management is important for maintaining the structure of the programming environment and supporting version control. The environment should also support cross-compatibility between different development environments, allowing for the transformation of projects into traditional text-based formats or REPL environments. This flexibility ensures that developers can switch between different tools as needed without losing progress.

Finally, the environment should offer extensibility and customization, enabling users to tailor their workspace and extend functionality as needed.

3.2 Features

Based on the aforementioned requirements, we derive the following features and prioritization for the design of IGC.

Incremental and Exploratory Programming. Incremental programming features, such as individual node execution, are core to IGC. The evolution of a program (and program) state is to be achieved using (small) code fragments. This feature will allow developers to test and refine code in manageable increments resulting in an efficient development process. To enable exploration, and inspired by the exploratory programming protocol of [9, 26], we add the ability to revisit a previously encountered program state and execute an alternative next code fragment. The various execution paths should remain available and made visual to the programmer. A graph of reached program states is gradually built, allowing the programmer to compare and contrast the effects of executions.

Complexity Management. This feature ensures that IGC can handle large, complex projects while maintaining scalability and readability. It directly stems from the advantages of text-based environments. It includes managing dependencies, tracking changes, and organizing code effectively to handle the complexity of large-scale software projects.

Code Architectural Visualization. Utilizing a graph structure to manage and visualize code relationships, enhancing project readability and maintainability. This requirement supports different architectural views, enabling developers to visualize code structures and relationships in various forms.

File Management. Efficiently storing and organizing the programming environment's structure within files, supporting other requirements like version control. This feature is crucial for maintaining a logical structure for projects and leveraging the full capabilities of version control systems.

The next section dives into our prototype IGC, describing how a subset of the above features and requirements are met by the current prototype.

4 Prototype

This section details the system architecture and some implementation choices of our IGC prototype, implemented as a web application. We describe both the front end and back end components. For our initial exploration we support Python as an object language because of its natural support for incremental programming (through its interpreter(s)) and extensive use in real-world projects. However, our goal is to support a wider range of programming languages and the architecture of Figure 2 has been designed with this aspect in mind.

4.1 React Web Application (Front End)

The front end of IGC is written in React with Typescript [8]. The front end consists of three main components: the file navigator, the graph editor, and the code editor.

The file navigator, located on the left side of the application (the left half of Figure 4), is responsible for file management.

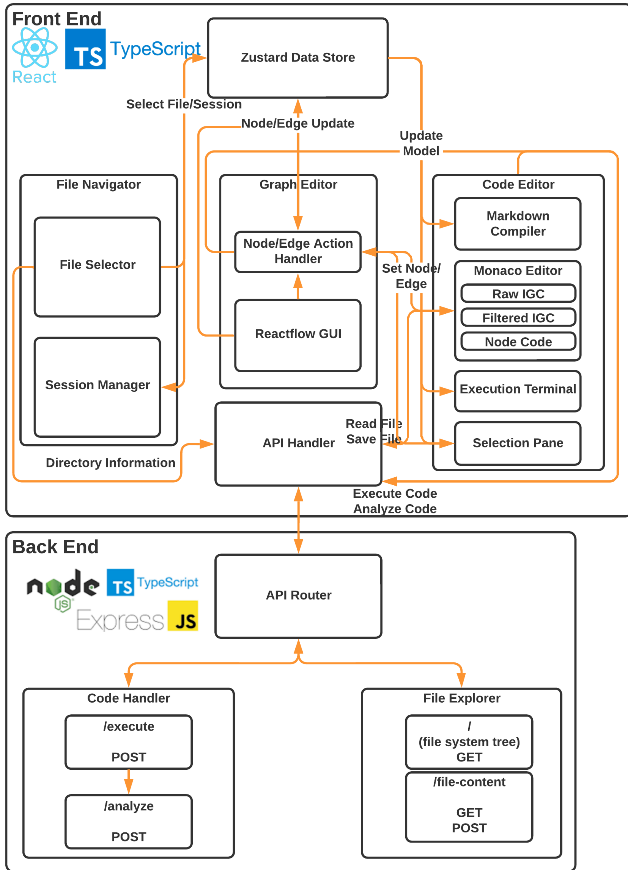

Figure 2. Application Diagram of IGC depicting the interactions between components of both the front and back end.

Users can select a file to display in both the graph editor and/or the code editor. At the bottom section of the file navigator, the session manager allows users to create, manage, and view sessions. A session tracks code execution, enabling users to switch between different sessions easily and export them for others to use.

In the middle of the application, the graph editor (the main left component in Figure 1) visualizes and edits the graph structure of an IGC file. The graph structure is managed by the ReactFlow[6] library, chosen for its performance, scalability, and customizability. The graph consists of nodes and directed edges called relationships.

There are currently seven types of nodes: Base, Class, Abstract Class, Interface, Method, Code Fragment, and Documentation. Each node type has unique properties and meanings, but all share the basic building blocks of code representation and analysis. For instance, a method node seeks a connected class node to attach to. Each node category offers different templates for user convenience, though users can also write their own code. The Code Fragment node serves general coding purposes, while the Documentation node stores and compiles markdown for display by other nodes. A special, automatically generated Start node represents the initial null execution state and begins all execution paths.

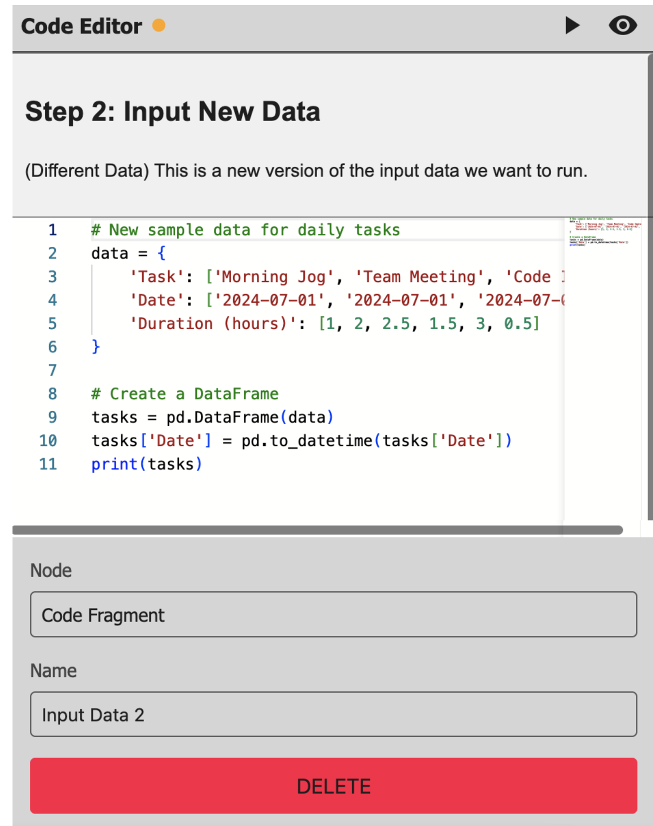

Figure 3. IGC Code Editor showing what would happen if you clicked a code fragment node. Any documentation would appear first, then the code, then general settings associated with the node. If the node has been executed, runtime information would also appear.

The graph has seven types of relationships: Base, Inheritance, Override, Method, Dependency, Execution, and Documentation. Each relationship type serves a specific purpose, either connected manually by the user, automatically based on code analysis, or through external interactions such as execution. Users drag from the dark grey bottom of the source node to the target node to create an edge. Custom path calculations prevent different edges from overlapping.

On the right side is the code editor (Figure 3), which displays the file's raw data selected in the file navigator. The primary code editor used is the Monaco React Editor[7], maintained by the team behind the VSCode editor. If an IGC file is selected, additional components may appear depending on the situation. A filtered version of the ReactFlow data appears in the editor by default. When a node is selected,

[6] https://reactflow.dev/

[7] https://microsoft.github.io/monaco-editor/

its associated code and any connected documentation nodes above the code are displayed. A selection pane at the bottom allows users to change the node type and name or, if an edge is selected, to change the relationship type. When a node is executed, additional options specific to the execution are shown, including stdout and stderr terminals to display output values, a configurations data pane showing the current configuration after execution, and a metrics pane with information such as execution time.

4.2 Node.js API (Back End)

The back end of our web application is built on a Node.js server, coded in TypeScript. Node.js was chosen for the back end server due to its seamless integration capabilities with a VSCode extension. However, a web application was preferred over a VSCode extension to allow greater control over various functionalities, which might be more challenging to achieve within the constraints of an extension environment. The back end server comprises two main routes: "file-explorer" and "code-handler."

The file-explorer route handles all file operations, including reading file trees, accessing specific file data, and saving specific file data. This route ensures efficient and organized file management, enabling users to navigate and manipulate their files with ease.

The code-handler route is responsible for code execution and analysis. The code execution API accepts raw code, the programming language (currently only Python is supported), and the current session. The process begins with code analysis to identify all newly defined variables, functions, modules, etc. If a state file exists from the session, it is loaded next. The actual code is then executed, and the resulting state is saved and compressed into a pickle file using Dill. All execution data is subsequently passed back to the front end.

Code analysis is managed by the AST Python library. The raw code is parsed to construct an AST, and nodes are visited to identify any dependencies required for code execution and any newly defined variables, functions, modules, etc., from the output of the raw code. This comprehensive analysis ensures that the back end can accurately track and manage the state of the code throughout its execution.

5 Showcase

The following showcase will demonstrate the functionality of IGC and compares IGC to the popular Jupyter Notebook environment. For the showcase, we create a program that helps track daily tasks, summarizes time spent, and provides simple suggestions for better task management.

Project setup. To get started, we first need to create a new project. This can be done by going to File (on the navigation bar) -> New Project. We are then left with a brand-new project environment. Let us create a new IGC file named "task_tracker.igc." The file is automatically populated with a node representing the default state of program execution. In other words, the node represents the initial state of any execution path.

Code and Documentation. To start development, we create a new node by clicking the "add node" button. There should be a new node on the graph. We change the type of the node to "Code Fragment," as we just plan to run the code as it is. The plan is to represent each node as it would be in a cell usually seen in a Jupyter Notebook. First, we would like to import some libraries needed for the rest of the script, so we change the node's name to "Import Libraries" and add the raw import statements to the node. Next, we add some documentation for the node in the form of markdown (as seen in many Notebooks) by double-clicking above the code cell next to the "+" symbol. This action creates a new Documentation cell attached to the code cell created earlier. We insert the markdown click on the original cell to place focus at that node. The markdown is subsequently compiled and shown. We repeat these steps to create a number of code cells, resulting in a structure resembling a Notebook written in the literal programming style (see Figure 4).

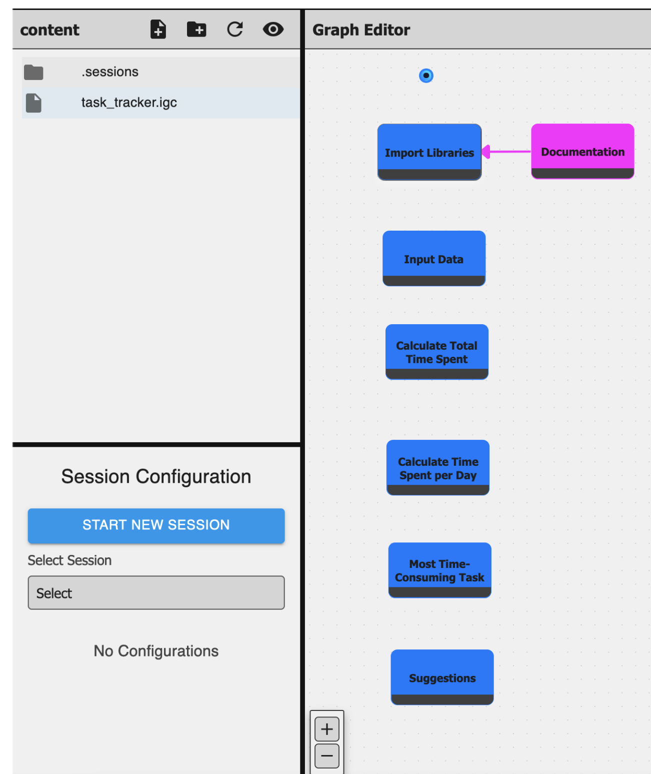

Figure 4. All code fragment nodes created visualization from the showcase. As the first (top) code fragment node is selected, a documentation node is present to show the corresponding documentation.

Building a session. Now that we have all the nodes created, let us try executing them. In theory, we could have executed each node after creation (and this would be more in line with the incremental programming philosophy). We decided to show the execution functionality afterward as we wanted to show a distinction between the creation of the cells and execution for demonstration purposes. First, we create a new session by clicking the "Start new session" button in the Session Configuration pane. To execute a node, we press the play button at the top of the code editor when a node is selected, or right-click the node and press run. We should now see an edge representing an execution relationship from the start node to the cell we executed. We also see a terminal with four tabs above it: "Output," representing the stdout; "Error," representing stderr; "Configuration," representing the newly defined state from this specific execution; and "Metrics," which shows metrics over the execution such as execution time. We should also see that the session configuration data has changed to the current state of the session execution. If we execute the next node, we see that we are creating the execution path leading from the previously executed node to the current node. With each execution, the session configuration changes to represent the current state of the execution. By clicking a previous node on the execution path we can inspect program state by clicking the 'configuration' tab of the node (see Figure 5).

Session Exploration. To experiment with modified input data, we create a new session by clicking the "Start new session" button. The execution edges from the previous session are no longer visible, and we can start defining a new execution path going into the new input data node. If we quickly want to compare the sessions, we can go to the session selector, and we see both sessions exactly in the state where we left them (Figure 6). Our showcase demonstrates how IGC has separated the concerns of establishing a narrative with code (and its documentation) – which is achieved through visual layout – and the alternative narrative of the execution of the code – which is achieved through maintaining multiple execution sessions. In this way, IGC addresses a key limitation in Jupyter Notebooks as described by [12, 14] from the perspective of exploratory programming.

6 Discussion

The following expands on the insights of the showcase of the previous section and discusses their implications.

Modular and Incremental Development. IGC retains the interactive and iterative nature of incremental programming while offering a 2-dimensional visualization. By breaking down code into distinct nodes, each representing a specific functionality or code fragment, IGC allows for modular development. Developers can isolate and test individual nodes in an isolated session before integrating them into a larger session. Developers can also create markdown text for explanations and tutorials. IGC can execute nodes independently and observe their outputs in isolation, which covers all the basic functionality of Jupyter's cell-based execution.

The main limitation compared to Notebooks is that IGC cannot yet create visual renderings of output. Another limitation is that code of a node is only displayed if the node is selected. In a Notebook all code cells are displayed simultaneously (although in sizeable Notebooks cell contents will span multiple screens). A feature currently in development is code projection to allow users to select one node and reveal all connected logic existing in other code fragments based on their dependencies.

Session Management. When a user (the reader) opens a Jupyter Notebook written by another user (the author), they are presented with a linear view of cells (the code narrative). The convention, promoting reproducibility, is to execute the cells in the order they are laid out. That is, if both the author and the reader follow this convention, then both users should[8] get the same results. However, this convention is not automatically upheld and is not always followed. Perhaps the author was only satisfied with the result after some experimentation, during which they executed cells in an order diverging from the code narrative. But the execution order is not reflected in the data format in which a Jupyter Notebook is exported. On the contract, in IGC, the user imports sessions alongside the code. Importing a project directly shows the order in which execution was performed by the author. Although this may be costly in terms of memory, users are also able to export and import sessions such that even configuration and output data can be shared (not just execution edges). Most importantly, multiple sessions can be developed and explored in IGC in parallel and the user is able to jump back-and-forth between sessions and program states.

IGC is highly flexible in terms of the supported interactions with node executions. Figure 7 is meant to demonstrate an outlandish execution path that IGC can handle without problem. This figure also reveals a difficulty with regards to the visual layout of edges. To avoid overlap between edges, a custom pathing algorithm was developed. However, realizing this goal in general is an NP-hard problem [6], and, for some graphs, is even undecidable [18].

Code Visualization. The graph editor in IGC serves as a tool for visualizing the structure of the code of a software project. By representing code components as nodes and their relationships as edges, developers gain a clear understanding of the project's structure. This visualization aids in identifying dependencies and understanding the overall flow of the application. The ability to see different types of nodes and relationships, such as class structures, executions, and dependencies, provides a complete view of the project, which can

[8] Although the Jupyter and Kernel versions possibly affect the outcome.

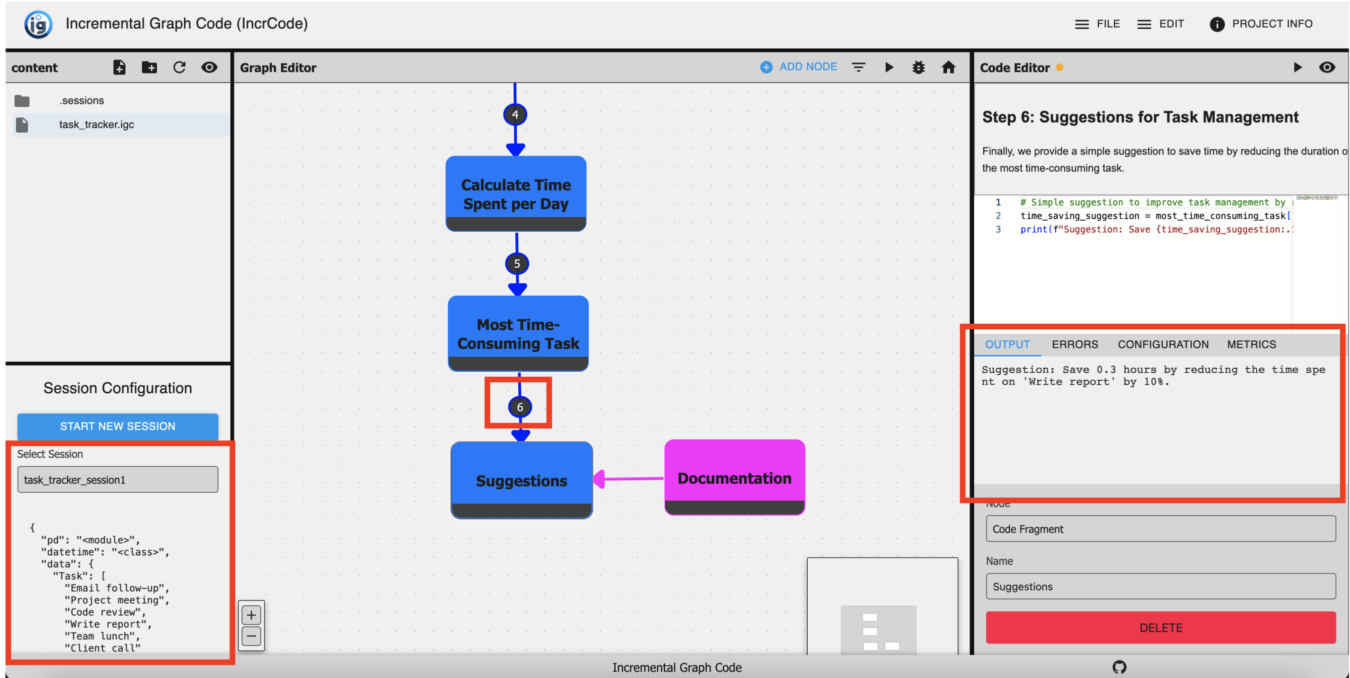

Figure 5. All code fragment nodes are executed visualization from the showcase. We also show the file navigator pane to display the updated session configuration resulting from the execution. After each execution, the execution label of the execution relationship increments. On the right side, the code editor displays the runtime data from the selected node (in this case, the last (bottom) node).

help to manage complexity. A potential point of future work is to integrate UML(-like) nodes and relations for building architectural designs from which code can be automatically generated, akin to development in a 'low-code' platform.

Scalability. A goal for IGC is to be able to handle large software projects, so scalability is a key consideration for its development. React was chosen as its virtual DOM and re-renders only updated components, ensuring smooth performance as the graph scales. ReactFlow can take advantage of React to optimize rendering.

ReactFlow provides an infinite, pannable workspace to create the graphs. This allows the user to create any size project, as the workspace can always be extended. A limitation is that to capture the entirety of a large project, the user must zoom out, which will eventually make text hard to read and elements difficult to see or distinguish. A possible solution currently in development is to consolidate subgraphs into a node by introducing a node type that can represent a (sub-)graph. This feature would allow users to simplify graphs by effectively introducing layers of abstraction. The sub-graph can be defined in a separate file, corresponding to the common development practice in which files often represent an individual component of a larger software project.

Zustand is used to record program state as it is lightweight, minimizes overhead and maintains consistency. Node.js is used for the back-end and is seen as highly scalable [25] owing to its non-blocking, event-driven architecture.

7 Future Work

This paper marks a point in an on-going investigation into both the user-facing UI elements and the underlying features that can potentially unite the benefits of incremental and exploratory programming with the needs of large-scale software development. The showcase given in this paper demonstrates the potential of our research, but thorough empirical evaluation is required. The following paragraph describes future research we intend to pursue. Later paragraphs describe technical extensions we envision for IGC to further achieve our goal, especially related to complexity management, and general usability improvements.

Empirical Validation. Future research will be focused on two main methods for empirical validation: evaluating usability with user studies and executing case studies to demonstrate the genericity and extensibility of the conceptual model underneath IGC. The goal of the usability study is to discover the extent to which our prototype succeeds in combining the benefits of incremental and exploratory programming with managing the complexity of large-scale software. In this study, a representative set of potential users will interact with the prototype and share their experiences

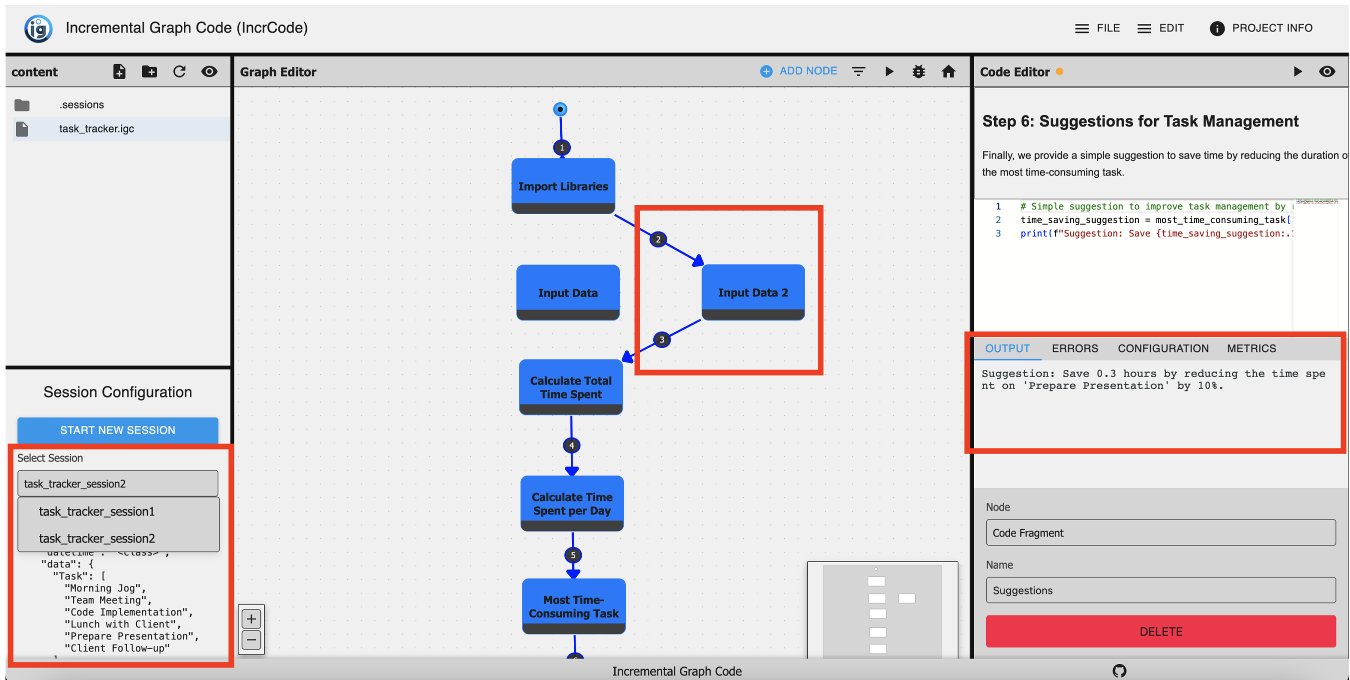

Figure 6. An alternate execution path is created in a new session. This demonstrates the effects of a new input code fragment node. The user can compare sessions by selecting different sessions in the session configuration pane located at the bottom of the navigation pane.

through interviews or surveys in a combination of quantitative and qualitative evaluation.

Another goal of our research has been to develop a conceptual model around a graph structure for storing and visualizing code that is extensible through the addition of new node types and edge types. In this conceptual design, new node- and edge-types are packaged with new views and controller functions on top of the graph, instantiating the Model-View-Controller (MVC) architectural pattern. To demonstrate the extensibility and generality of our conceptual model, we intend to describe a number of extensions in a modular fashion and demonstrate their ability to (collectively) realize features encountered in other novel programming environments or (Jupyter) Notebook extensions. Particular targets for case studies are (UML-)diagrams of low-code platforms, managing complexity with architectural patterns such as MVC, data science pipelines in Link by MakinaRocks[9], and documentation management in PescaJ [19].

Importing Different IGC Files. The ability to import different IGC files into a project or application is currently under development. This feature will allow users to consolidate groups of code into components, enhancing modularity and reusability. By enabling the integration of multiple IGC files, developers can build more complex applications efficiently, leveraging previously created components.

Code Projection. There is currently an observability limitation with having a graph structure. Only the node that is selected will be displayed in the code editor. Code projection is another feature under development that will allow users to quickly see all connected code fragments to the one they select. This connection could represent several views, such as class-centric views, relationship views, etc. Currently, only documentation projections are incorporated in IGC. This means that selecting a code node will automatically display the corresponding documentation (if it exists).

Source Control. Implementing source control in IGC is a high priority. Integrating a versioning approach similar to Variolite [13] would align well with the exploratory programming philosophy of IGC, allowing for seamless tracking and management of different code versions. Traditional text-based diffing methods like the Myers Algorithm can be used for node contents, with graph-diffing algorithms, such as NodeGit [2], for the overarching graph structure.

Export and Import Functionality. To facilitate a smoother transition for users, it is essential to implement functionality that allows for exporting projects to REPL or text-based environments and vice versa. This capability would enable developers to move their work between different environments easily, helping them get accustomed to IGC without disrupting their existing workflows.

[9] https://link.makinarocks.ai/

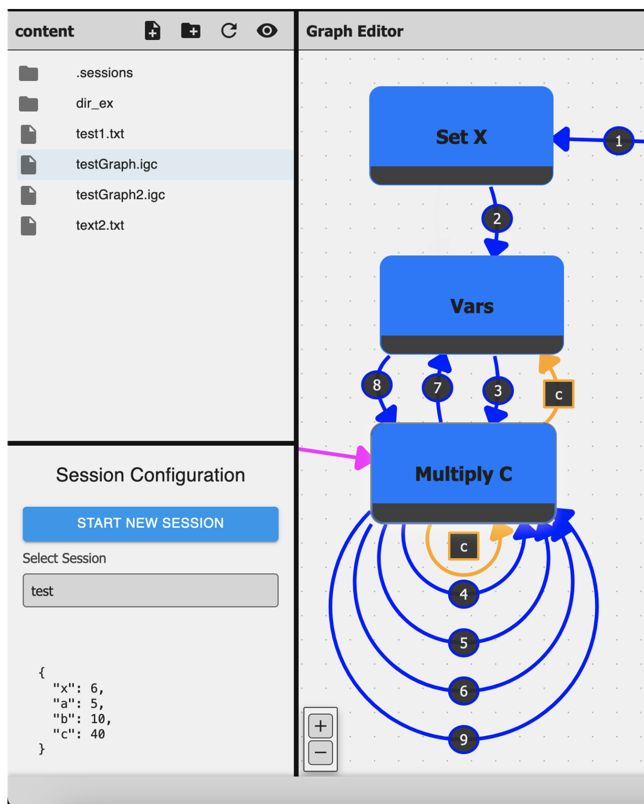

Figure 7. A demonstration of a complex execution path. The execution is switching between various code nodes.

Extensibility. A significant goal for IGC is to foster a community-driven ecosystem where users can create and share extensions and features. To achieve this, IGC should support the development of custom nodes and relationships, allowing users to represent various programming concepts flexibly. Implementing an add-on system would empower users to extend the environment's functionality, making IGC a versatile and adaptable tool for diverse programming needs.

Exploratory Programming Enhancements. Exploratory programming is a major consideration in IGC. The sessions functionality has enabled users to experiment with different execution paths, however, this can be enhanced by giving a better overview of all sessions. Expanding on sessions, they are currently limited to starting from the initial execution state every time a new session is created. An upcoming feature will be to create a new execution branch directly off of a pre-existing session at any point of time. This will allow users to more easily manipulate and experiment with different code fragments.

8 Related Work

This section compares IGC with related work.

Terminal-Based REPLs. Terminal-based REPLs offer a linear, command-line interface where users input code line by line, receive immediate feedback, and manage state sequentially. In contrast, IGC provides a graphical interface with a visual node-based editor, allowing users to interact with code more intuitively, manage multiple sessions, and visualize complex code structures and relationships.

Model-Driven Development. The more abstraction that can be done to resemble design patterns, the more this approach seems to imitate the goals of Model-Driven Development (MDD). MDD is a software development approach that focuses on creating and utilizing abstract models. MDD prioritizes the design aspect of the software, using models as simplified representations to guide code generation and other development processes [21]. However, the consequences of a model-first approach in MDD include challenges like redundancy in managing multiple model representations, difficulties in formalizing a standard modeling language, and increased complexity in managing model relationships [10]. IGC differentiates itself from MDD by offering a classical coding-first approach through REPLs that can later be abstracted to design patterns instead of the other way around.

Visual Programming Languages. Visual Programming Languages (VPLs), like MDD, have many overlaps with the aspirations of IGC. VPLs are about transforming visual representations into program logic. One of the most relevant subsets of VPLs is called Node Graph Architecture (NGA), which is about representing code statements into nodes of a graph. This architecture is at the extreme end of the visualization and abstraction of what IGC is trying to accomplish. IGC differs as it is meant to be a hybrid environment between classical text development. However, one can not understate the relevance of VPLs.

PescaJ. PescaJ [19], presented at PAINT in 2023, is a projectional editor designed for Java that addresses the challenge of scattered code and documentation through aggregated views. PescaJ offers a departure from conventional text-based editing by allowing developers to create customizable, overlapping views that consolidate both code and documentation fragments, which are often dispersed across different files and classes in traditional IDEs. Once a code fragment or documentation is selected, related code or documentation is displayed next to it. Currently, IGC tries to incorporate code projections by displaying corresponding documentation nodes. Currently, a goal is to expand this feature to provide similar visuals, specifically regarding code fragments throughout the graph.

Alternative Code Structures. Other works have investigated alternative ways of structuring and visualizing code rather than relying on a file-system (in an IDE) or a linear sequence of code cells (in a Notebook). Code Bubbles affords the visual organization of code fragments (as 'code bubbles')

across the working pane and enables grouping bubbles in various (visual) ways. The evaluation shows that the approach has the potential to help users with a wide range of development tasks, including reading, editing, and navigating source code, as well as supporting multitasking and breakpoint debugging [4]. In particular, users have been shown to spend significantly less time navigating when performing tasks related to understanding code [5].

The Link environment by MakinaRocks[10] adds a visual display to Jupyter Notebooks showing a graph representation of the notebook's code cells. The code cells are shown as pipeline components (nodes) and are connected via dependencies (edges). Users can click on a node to request execution of the code cell and the code cells it depends on (in the order respecting the dependencies). The visual pipeline helps users track and manage the evolution of their code, but alternative execution paths cannot be simultaneously explored.

Harden et al. provide an investigation into the design and evaluation of the potential of Notebooks with a 2D cell layout [11]. The 2D layout affords non-linear code narratives and admits a form of branching not supported by conventional Notebooks. Their user studies reveal that users indeed take advantage of the 2D structure and that the layout promotes exploratory programming. This demonstrates the ability to branch analyses and to easily compare results.

Observable. The Observable Notebook is an IPE computational Notebook for creating and sharing live, reactive data visualizations and analyses. Unlike traditional Notebooks, Observable uses a reactive programming model, meaning that each cell in the Notebook automatically updates whenever its dependencies change. This real-time reactivity is achieved through a system where cells can refer to each other by name, and any updates in one cell propagate instantly to others that depend on it. Although an interesting feature, IGC does not incorporate this reactive model as the main extension of IPEs that IGC hopes to accomplish is complexity management. While the reactive model could be beneficial to prevent micromanaging of sessions, it introduces many considerations that must be addressed such as cyclic execution.

9 Conclusion

This paper has described a novel approach for building programming environments based on an internal graph structure to represent code fragments and various types of relations between code fragments (e.g., structural relations, execution order, and dependencies). The IGC prototype presented in this paper is part of a larger research effort investigating programming environments that attempt to bring together the distinctive features of REPLs, Computational Notebooks, and Integrated Development Environments such as incremental programming, exploratory programming and managing complex source code. This way, different programming styles and projects can be supported within the same programming environment.

IGC improves complexity management through its visualization of code fragments and their relationships. The visual organization of code not only improves readability but also allows for effective handling of dependencies and project scalability, which is often a challenge in REPLs and Jupyter Notebooks. The modular nature of IGC supports incremental development, enabling developers to isolate, test, and integrate code fragments with little effort.

In a showcase, we demonstrated that IGC can support Notebook-style programming with additional features for exploratory programming. Developers can manipulate execution paths to explore different scenarios and compare outputs easily. Immediate feedback is provided which is key to understanding code behavior and causality at development time. The added flexibility improves experimentation compared to conventional Jupyter Notebooks.

Although the initial experiments with IGC are promising, demonstrating the full strength of the suggested approach requires adding new features and the execution of a number of case studies as future work. In particular, we aim to add a node type for representing sub-graphs, further enhancing the ability to manage complex projects. We aim to execute a case study involving a large software project that implements a Model-View-Controller (MVC) architecture in which the source code is visually organized in our environment according to the MVC distinction.

[10] https://link.makinarocks.ai/

References

[1] Mary Beth Kery and Brad A. Myers. 2017. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 25–29. https://doi.org/10.1109/VLHCC.2017.8103446 ISSN: 1943-6106.

[2] Marcel Borowski, Johannes Zagermann, Clemens N. Klokmose, Harald Reiterer, and Roman Rädle. 2020. Exploring the Benefits and Barriers of Using Computational Notebooks for Collaborative Programming Assignments. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20)*. Association for Computing Machinery, New York, NY, USA, 468–474. https://doi.org/10.1145/3328778.3366887

[3] Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, and Benoit Baudry. 2018. Omniscient debugging for executable DSLs. *Journal of Systems and Software* 137 (March 2018), 261–288. https://doi.org/10.1016/j.jss.2017.11.025

[4] 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 *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 455–464. https://doi.org/10.1145/1806799.1806866

[5] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. 2010. Code bubbles: a working set-based

interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. Association for Computing Machinery, New York, NY, USA, 2503–2512. https://doi.org/10.1145/1753326.1753706

[6] Sergio Cabello and Bojan Mohar. 2013. Adding One Edge to Planar Graphs Makes Crossing Number and 1-Planarity Hard. *SIAM J. Comput.* 42, 5 (2013), 1803–1829. https://doi.org/10.1137/120872310

[7] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. 2020. What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3313831.3376729

[8] Boris Cherny. 2019. *Programming TypeScript: Making Your JavaScript Applications Scale*. "O'Reilly Media, Inc.". ISBN: 978-1-4920-3762-0.

[9] Damian Frolich and L. Thomas van Binsbergen. 2021. A Generic Back-End for Exploratory Programming. In *Trends in Functional Programming*, Viktória Zsók and John Hughes (Eds.). Springer International Publishing, Cham, 24–43. https://doi.org/10.1007/978-3-030-83978-9_2

[10] B. Hailpern and P. Tarr. 2006. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal* 45, 3 (2006), 451–461. https://doi.org/10.1147/sj.453.0451 Conference Name: IBM Systems Journal.

[11] Jesse Harden, Elizabeth Christman, Nurit Kirshenbaum, John Wenskovitch, Jason Leigh, and Chris North. 2022. Exploring Organization of Computational Notebook Cells in 2D Space. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–6. https://doi.org/10.1109/VL/HCC53370.2022.9833128 ISSN: 1943-6106.

[12] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300500

[13] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. Association for Computing Machinery, New York, NY, USA, 1265–1276. https://doi.org/10.1145/3025453.3025626

[14] Mary Beth Kery and Brad A. Myers. 2018. Interactions for Untangling Messy History in a Computational Notebook. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 147–155. https://doi.org/10.1109/VLHCC.2018.8506576 ISSN: 1943-6106.

[15] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3173574.3173748

[16] Thomas Kluyver, Benjamin Ragan-Kelley, Pé, Fernando Rez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damiá Avila, n, Safia Abdalla, Carol Willing, and Jupyter Development Team. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. IOS Press, 87–90. https://doi.org/10.3233/978-1-61499-649-1-87

[17] D. E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (Jan. 1984), 97–111. https://doi.org/10.1093/comjnl/27.2.97

[18] O. Levin. 2018. *Discrete Mathematics: An Open Introduction*. Amazon Digital Services LLC - Kdp. https://books.google.nl/books?id=YTAWwQEACAAJ ISBN: 978-1-79290-169-0.

[19] José Lopes and André 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 2023)*. Association for Computing Machinery, New York, NY, USA, 44–50. https://doi.org/10.1145/3623504.3623571

[20] Kurt Nørmark. 2009. Systematic Unit Testing in a Read-eval-print Loop. (Oct. 2009). https://doi.org/10.3217/jucs-016-02-0296

[21] Oscar Pastor, Sergio España, José Ignacio Panach, and Nathalie Aquino. 2008. Model-Driven Development. *Informatik-Spektrum* 31, 5 (Oct. 2008), 394–407. https://doi.org/10.1007/s00287-008-0275-8

[22] Jeffrey M. Perkel. 2018. Why Jupyter is data scientists' computational notebook of choice. *Nature* 563, 7729 (Oct. 2018), 145–146. https://doi.org/10.1038/d41586-018-07196-1

[23] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and Live, Programming and Coding. *The Art, Science, and Engineering of Programming* 3, 1 (July 2018), 1:1–1:33. https://doi.org/10.22152/programming-journal.org/2019/3/1 Publisher: AOSA, Inc..

[24] Yahya Tashtoush, Noor Abu-El-Rub, Omar Darwish, Shorouq Al-Eidi, Dirar Darweesh, and Ola Karajeh. 2023. A Notional Understanding of the Relationship between Code Readability and Software Complexity. *Information* 14, 2 (Feb. 2023), 81. https://doi.org/10.3390/info14020081 Number: 2 Publisher: Multidisciplinary Digital Publishing Institute.

[25] Pedro Teixeira. 2012. *Professional Node. js: Building Javascript based scalable software*. John Wiley & Sons. ISBN: 978-1118185469.

[26] L. Thomas van Binsbergen, Damian Frölich, Mauricio Verano Merino, Joey Lai, Pierre Jeanjean, Tijs van der Storm, Benoit Combemale, and Olivier Barais. 2022. A Language-Parametric Approach to Exploratory Programming Environments. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2022)*. Association for Computing Machinery, New York, NY, USA, 175–188. https://doi.org/10.1145/3567512.3567527

[27] L. Thomas van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs van der Storm, Benoit Combemale, and Olivier Barais. 2020. A principled approach to REPL interpreters. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2020)*. Association for Computing Machinery, New York, NY, USA, 84–100. https://doi.org/10.1145/3426428.3426917