Coding through Storytelling: Narrative Reasoning and Software Engineering Education

S. Charlie Dey
Texas Advanced Computing Center
fcharlie@tacc.utexas.edu

Jeaime H. Powell
Omnibond Systems
fjeaime@omnibond.com

Victor Eijkhout
Texas Advanced Computing Center
feijkhout@tacc.utexas.edu

Joshua Freeze
Texas Advanced Computing Center
fjfreeze@tacc.utexas.edu

Susan Lindsey
Texas Advanced Computing Center
fslindsey@tacc.utexas.edu

ABSTRACT

To become a successful software engineer, technical competence alone is not enough. Students must learn to reason about their code, articulate their intentions, and locate errors with clarity and confidence. This paper introduces a pedagogical approach rooted in the metaphor of "telling a story." By encouraging students to narrate their code—identifying protagonists (variables), plotlines (control flow), and conclusions (outputs)—we promote a practice of self-explanation that strengthens metacognitive awareness and debugging skills. Drawing from experiences in the classroom, we show how storytelling helps students pinpoint bugs, communicate intent, and ultimately write more understandable code. We connect these practices with existing research on metacognition, program comprehension, and human-centered computing, and describe how this narrative approach provides a scalable, inclusive, and transferable tool for future computational engineers and scientists.

KEYWORDS

Metacognition, Human-centered programming, teaching coding

1 INTRODUCTION

In undergraduate high-performance computing (HPC) programming courses, a common challenge is students' difficulty in articulating the context of their code during debugging sessions. Typically, students focus narrowly on specific syntactic, semantic, or logical errors, omitting the broader purpose of their program, which hinders effective instructor guidance. Traditional prompts like "walk me through your code" often prove challenging as students simultaneously navigate the problem domain, programmatic issues, and the syntax of a new programming language. To address this, a pedagogical approach grounded in narrative reasoning—a cognitive framework that leverages humans' natural ability to process sequences and causality—has been adopted [2]. This method, implemented through a storytelling exercise, encourages students to shift from a micro-level focus on errors to a macro-level observation of their code's logical narrative, aligning with constructivist learning principles.

1.1 Classroom Implementation

The storytelling approach involves prompting students to "tell the story of your code," often with a playful opener like "Once upon a time…" This shifts their perspective from being immersed in technical details to viewing their code as a narrative with characters (variables), plot (control structures), and resolution (outputs). For example, a student might describe a variable as a character navigating a dataset, making decisions based on conditions, and collecting results. When the narrative falters—such as when a student cannot logically continue the story—it often signals a misunderstanding of a function, algorithm, or logical flow, pinpointing the bug's location. This process also reveals syntactic errors, like missing punctuation or misnamed variables, as students verbalize their logic in a human-readable format.

This paper argues that narrative reasoning—a natural human faculty for explaining sequences, causality, and change—can be harnessed as a powerful tool in software engineering education. Specifically, it supports:

- Self-explanation during coding and debugging
- Improved program comprehension through structured reasoning
- Communication and collaboration via shared narrative framing

Through classroom observations, existing research, and proposed curricular strategies, we introduce a narrative framework that promotes reflective practice among novice programmers.

2 DISCUSSION

Grounded in constructivist learning theory, the storytelling approach encourages learners to actively build mental models by narrating the structure and intent of their code. We present a model that maps programming constructs to narrative elements—such as characters, setting, and plot—to promote comprehension, self-explanation, and reflective practice. Supported by research in metacognition, program comprehension, and learner-centered design, this approach not only improves debugging and problem-solving skills but also fosters engagement and communication, particularly among novice and diverse learners. Together, these foundations establish storytelling as both an effective teaching tool

https://doi.org/10.22369/issn.2153-4136/16/2/2

and a cognitive scaffold for navigating the complexities of programming.

2.1 Educational Framework: Constructivism

This approach is rooted in the constructivist learning framework, which posits that learners actively construct knowledge through experience and reflection [9]. By narrating their code, students externalize their mental models, making assumptions and errors explicit, which facilitates refinement of their understanding. The narrative structure serves as a scaffold, connecting new programming concepts to familiar storytelling patterns, thus enhancing comprehension and retention.

2.2 Model Overview

The educational model for "coding through storytelling" is a narrative pedagogy that encourages students to conceptualize code as a story with narrative elements: characters (variables and objects), plot (control flow), setting (initial conditions), conflict (conditionals or loops), and resolution (outputs or goals). Grounded in constructivism, which posits that learners actively construct knowledge through experience and reflection [9], this model integrates several pedagogical practices to foster metacognition, problem-solving, and communication skills. It aligns with research on program comprehension as storytelling [2] and metacognition in programming [5].

2.3 Benefits of the Storytelling Approach

The narrative-based pedagogy offers several educational benefits, supported by empirical research:

(1) Improved Debugging: Narrating code shifts students' focus to the program's logical flow, revealing inconsistencies or misunderstandings that pinpoint bugs. This aligns with findings on self-explanation, which show that verbalizing thought processes enhances problem-solving [8].
(2) Enhanced Metacognition: The act of storytelling fosters metacognitive awareness, encouraging students to reflect on their understanding and problem-solving strategies, a critical skill in programming education [5].
(3) Increased Engagement: The playful prompt "Once upon a time…" adds levity to debugging, making it more approachable, particularly for diverse learners, as supported by learner-centered computing education principles [4].
(4) Better Communication: Narrating code helps students articulate their intent clearly, a vital skill for collaborative software development, aligning with broader educational goals [4].
(5) Enhanced Program Comprehension: Framing code as a narrative improves understanding by leveraging natural cognitive tendencies to process stories, as evidenced by research on program comprehension [2].

2.4 Academic Foundations

The storytelling approach is supported by several areas of computing education research:

- Narrative Reasoning: A systematic review highlights storytelling's role in software development, noting its benefits in planning, requirements elicitation, and prototyping, providing a theoretical basis for its use in education [2].
- Self-Explanation: Verbalizing thought processes during programming improves learning outcomes, as demonstrated in a randomized experiment with self-explanation assignments [8].
- Metacognition: Metacognitive strategies, such as reflection and self-regulation, are essential for programming novices, and storytelling prompts such reflection [5].
- Program Comprehension: Understanding code as a narrative enhances comprehension by aligning with cognitive processes for processing sequences and causality [1].
- Learner-Centered Design: Storytelling makes programming more accessible and engaging, particularly for diverse student populations, aligning with learner-centered educational approaches [4].

2.5 Metacognition and Self-Explanation

Metacognition—thinking about one's own thinking—is a critical skill in learning to program. Students who engage in self-explanation while coding are more likely to detect misunderstandings and refine their mental models of how code behaves. Loksa et al. [5] describe self-regulation strategies in novice programmers, including planning, monitoring, and debugging as metacognitive acts that enhance learning outcomes.

Narration is one such form of self-explanation. By "telling the story" of what a program does, students externalize their reasoning process—making it easier to spot contradictions, misconceptions, or gaps in understanding.

2.6 Code as Story

Ciancarini et al. [2] explored the idea that program comprehension can be improved by treating code as a narrative structure. They liken variables to characters, control flow to plot, and comments to narration—a framing that helps learners understand not just what the code does, but why. This aligns with how expert programmers often read code: not linearly, but semantically—constructing a story that explains behavior.

Recent neuroscience also supports this framing. Peitek et al. [6] used fMRI to show how code complexity and vocabulary burden working memory during comprehension, aligning with storytelling's role in structuring mental load.

2.7 Debugging, Deconstruction, and Literacy

Debugging as hypothesis testing is a narrative act—imagining what should happen, then identifying where the story breaks. Griffin [3] advocates for "deconstructionist" learning, where reading, tracing, and debugging come before code writing, aligning well with our approach.

Storytelling in teaching also supports literacy, creativity, and critical thinking. Satriani [7] found storytelling enriched vocabulary, engagement, and comprehension in literacy classrooms—benefits that transfer to code comprehension as well.

3 THE NARRATIVE PEDAGOGY

3.1 Characters, Plot, and Purpose

In the classroom, we frame code as a story with:

- Protagonists: Variables, objects, or agents that change or act
- Setting: Initial conditions and inputs
- Plot: Control flow — how the story unfolds step-by-step
- Conflict: Conditionals or loops that create decision points
- Resolution: Output, state changes, or goals achieved

This structure helps students identify the logical and conceptual components of a program more intuitively. By mapping code to familiar storytelling elements, students create a mental schema that aids in both comprehension and retention.

We often begin exercises by giving students a blank narrative framework and asking them to fill it in with elements from a given code snippet or problem description. This reverse-engineering of story from code builds analytical skills, while designing stories before writing code builds synthesis.

3.2 Writing the Story Before the Code

While debugging often reveals where the narrative breaks down, equally powerful is the practice of writing the story first. Before any code is written, students are encouraged to map out the "story arc" of their program:

- Who are the characters? (variables, inputs, actors)
- What is the setting? (initial conditions or assumptions)
- What is the problem/conflict? (what needs to be solved or computed)
- What is the plot? (algorithm or logical steps)
- How does the story end? (output or goal)

This pre-coding narrative acts as a mental simulation of the algorithm and guides students away from immediately jumping into syntax. It shifts the focus from what code to write to what problem to solve—a critical reframe, especially for novice programmers.

This practice also aligns with professional software design principles, such as:

- Test-driven development, where the tests represent the expected outcomes of the story
- Design-first thinking, where logic is mapped out before implementation
- Algorithm sketching, used in pseudocode or flowchart form to visualize intent

By embedding storytelling at the design stage, we encourage foresight, structure, and intentionality in programming. It also supports better communication in team settings, as students can articulate the purpose and logic of their code before implementation.

3.3 Finding the Bug Through the Broken Story

In debugging exercises, students are asked to "read aloud the story" of their code. Where the story breaks—where they pause, contradict themselves, or say "I'm not sure what this part does"—is almost always the bug's location.

This technique externalizes cognition, making their mental model visible. It also mirrors professional practices: code reviews and pair programming sessions often revolve around shared narrative explanations of what code is doing.

We also incorporate peer-debugging activities where students exchange code and provide narrative explanations of what the code is "supposed" to do. This collaborative storytelling not only builds comprehension but fosters peer support and critical dialogue.

4 CLASSROOM IMPLEMENTATION

4.1 "Storytelling Debugging" Sessions

In small-group help sessions, office hours, or lab time, instructors and TAs engage students in live storytelling about their code. These sessions begin with the student walking through their program line by line, describing what each part is supposed to do, in their own words. Interruptions or breakdowns in the narrative usually signal areas of confusion or bugs. The instructor can then prompt deeper reflection with questions like:

- What's the role of this variable here?
- What happens next in the story?
- Does the ending make sense given the plot so far?

This Socratic approach encourages self-correction and metacognitive awareness. These sessions are particularly effective in early-course projects where logic may be simple, but the storytelling gap is often large.

4.2 Reflective Journals and Pair Storytelling

To reinforce the storytelling practice outside structured help sessions, students maintain reflective journals where they document the story of their code weekly. These can include narrative descriptions, flow diagrams, or even short paragraphs written as if explaining their solution to a non-programmer. Prompts might include:

- What was the goal of my code this week?
- Who were the main characters (variables/functions)?
- What surprised me in the process?

In "pair storytelling," students take turns reading and retelling each other's code in small groups. This not only strengthens their understanding of syntax and semantics but also develops critical peer review skills. When a peer can't explain a section clearly, it becomes a clue for the author to refine either the code or their internal logic.

4.3 Rubrics and Evaluation

Assessment in storytelling-based instruction can include narrative clarity as a rubric category. Instructors can evaluate:

- The completeness and coherence of a student's narrative explanation
- The alignment between the intended story and actual code behavior
- The student's ability to identify turning points or conflicts in the logic

These assessments don't replace functional correctness but enrich it, allowing instructors to gain deeper insight into student thinking and development.

4.4 Integration with Curriculum

The narrative framework is not a standalone technique—it can be scaffolded across the curriculum. Early weeks can introduce storytelling as a reflection tool. By mid-semester, storytelling becomes a design strategy before implementation. In later projects, students use it collaboratively for design proposals and team check-ins. Embedding narrative reasoning throughout ensures it's internalized as part of the student's cognitive toolkit.

4.5 Example Activities

Consider a student debugging a parallel computing program designed to process large datasets. When asked to explain their issue, they might initially point to a specific line causing a runtime error. By prompting them to "tell the story of your code," the instructor encourages a narrative: "Once upon a time, a variable named data_chunk set out to process a portion of the dataset. It entered a loop to compute averages, but then it got stuck because the loop never ended." This narrative might reveal that the student misunderstood the termination condition of the loop, highlighting a logical error. As the student continues, they might notice a missing semicolon that disrupted the syntax, which becomes apparent when explaining the code's flow. This example illustrates how storytelling shifts the student's perspective, enabling them to identify both logical and syntactic issues. Examples include:

- The Delivery Bot: Debugging broken logic by following the story of a package.
- Game Storyboarding: Mapping input, flow, and outcome in games like Rock-Paper-Scissors.
- Data Visualization Stories: Connecting data interpretation with narrative.
- Peer Storytelling: Retelling and debugging a partner's code to highlight clarity and intent.

4.5.1 Example 1: The Delivery Bot (Narrative Debugging). Students are given the following function:

```
def setDeliveryTime(package):
   if not package['priority'] and package['weight'] > 10:
        return "afternoon"
    else:
        return "morning"
```

They are asked to tell the story of what happens to each package:

```
package1 = {'priority': True, 'weight': 12}
package2 = {'priority': False, 'weight': 8}
package3 = {'priority': False, 'weight': 15}
```

They identify that a priority package might still go in the morning even if it's very heavy—an unintended outcome. This prompts a redesign:

```
def setDeliveryTime(package):
    if package['priority']:
        return "morning"
    elif package['weight'] <= 10:
        return "morning"
    else:
        return "afternoon"
```

4.5.2 Example 2: Game Mechanics as Story (Design-First Storyboarding). Before coding a Rock-Paper-Scissors game, students storyboard the interaction:

- Who are the players?
- What events trigger the next move?
- How does the system declare a winner?

Students present the story visually before implementing any logic. This helps identify branching conditions and user interaction flow.

4.5.3 Example 3: Data Storytelling with Visualization (Narrative Reflection). After analyzing a dataset and creating a visualization, students write a brief narrative explaining:

- What the data shows
- What question their code answers
- What story the visualization tells

This aligns narrative structure with data reasoning.

4.5.4 Example 4: Explaining a Peer's Code (Peer Storytelling). Students swap programs and narrate each other's code out loud. Prompts include:

- What is the goal of the program?
- What are the key steps along the way?
- Where is a potential flaw in the story?

This activity strengthens code readability, testing comprehension, and fosters collaborative debugging.

5 LIMITATIONS AND FUTURE WORK

While promising, narrative pedagogy has limitations. Students with less fluency in English or those more familiar with mathematical abstractions may find storytelling unnatural at first. Additionally, there is a risk of oversimplifying technical concepts if too much emphasis is placed on metaphor over precision. Instructors must strike a balance between promoting narrative clarity and ensuring computational correctness.

This approach also requires additional instructional time and scaffolding that may not be feasible in all classroom contexts. Larger courses may face challenges integrating personalized storytelling activities or giving feedback on reflective work like journals.

Another limitation is the potential variance in how students interpret and construct stories. While diversity of thought can be an asset, it can also lead to misaligned mental models if not guided carefully. Further research is needed to explore how cultural and linguistic backgrounds influence narrative construction in programming education.

Future work will investigate how narrative practices scale in larger courses, how they integrate with peer programming and automated assessment tools, and whether they foster long-term improvement in code quality and debugging efficiency. Controlled studies comparing narrative and non-narrative cohorts could validate impact on learning outcomes. There is also opportunity for tool development—intelligent IDEs or tutoring systems that can prompt students to articulate their story during code construction.

Moreover, the emergence of AI and large language models (LLMs) presents an interesting new frontier. Narrative-based pedagogy may

inform the development of smarter educational prompts, explainable AI code assistants, and curriculum-aware LLMs that can co-construct stories with learners. By studying how students tell code stories, we may also improve how machines understand, generate, and teach code narratives—paving the way for more collaborative, human-centered computing.

6 CONCLUSION

While promising, the storytelling approach faces challenges, such as objectively assessing narrative quality and scaling it to large classes, which requires instructor training. Some students may initially find narrating code awkward, particularly if they lack confidence. Future research could quantify the approach's impact through controlled studies and explore tools to integrate storytelling into programming environments, as suggested by related work [2].

Storytelling is a universal human tool for reasoning through complexity. By treating code as a story, students learn to explain their thinking, debug more effectively, and write clearer programs. This paper presented a narrative pedagogy that maps programming constructs to storytelling elements—variables as characters, control flow as plot, and output as resolution. Such framing supports student understanding, especially when reinforced through narrative-based planning and story-first design strategies outlined.

We explored classroom practices that support storytelling, including reflective journaling, peer code narration, and live debugging as story deconstruction. These methods empower students to identify and resolve logic gaps, communicate their intent, and grow into thoughtful engineers. As software becomes more embedded in every discipline, helping students become reflective, communicative, and narrative-driven coders prepares them not only for technical success but also for collaborative, human-centered innovation.

As software becomes more embedded in every discipline, helping students become reflective, communicative, and narrative-driven coders prepares them not only for technical success but also for collaborative, human-centered innovation.

REFERENCES

[1] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* 18(6) (1983), 543–554. https://doi.org/10.1016/S0020-7373(83)80031-5

[2] Paolo Ciancarini, Mirko Farina, Ozioma Okonicha, Marina Smirnova, and Giancarlo Succi. 2023. Software as storytelling: A systematic literature review. *Computer Science Review* 47 (2023), 113–120. https://doi.org/10.1016/j.cosrev.2022.100517

[3] Jean Griffin. 2016. Learning by taking apart: Deconstructing code by reading, tracing, and debugging. In *Proceedings of the 17th Annual Conference on Information Technology Education (SIGITE '16)*. Association for Computing Machinery, Boston, Massachusetts, 148–153. https://doi.org/10.1145/2978192.2978231

[4] Mark Guzdial. 2015. Learner-Centered Design of Computing Education: Research on Computing for Everyone. *Synthesis Lectures on Human-Centered Informatics* 8(6) (2015), 1–165. https://doi.org/10.2200/S00684ED1V01Y201511HCI033

[5] Dastyni Loksa, Lauren Margulieux, Brett A. Becker, Michelle Craig, Paul Denny, Raymond Pettit, and James Prather. 2022. Metacognition and self-regulation in programming education: Theories and exemplars of use. *ACM Transactions on Computing Education* 22(4) (2022), 1–37. https://doi.org/10.1145/3487050

[6] Norman Peitek, Sven Apel, Chris Parnin, and Andre Brechmannand Janet Siegmund. 2021. Program comprehension and code complexity metrics: An fMRI study. In *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. Madrid Spain, 524–536. https://doi.org/10.1109/ICSE43902.2021.00056.

[7] Intan Satriani. 2019. Storytelling in teaching literacy: Benefits and challenges. *English Review: Journal of English Education* 8 (2019), 113–120. https://doi.org/10.25134/erjee.v8i1.1924

[8] Arto Vihavainen, Craig S. Miller, and Amber Settle. 2015. Benefits of self-explanation in introductory programming. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. Association for Computing Machinery, Kansas City, Missouri, 284–289. https://doi.org/10.1145/2676723.2677260

[9] Ernst von Glasersfeld. 1989. Cognition, construction of knowledge, and teaching. *Synthese* 80 (1989), 121–140.