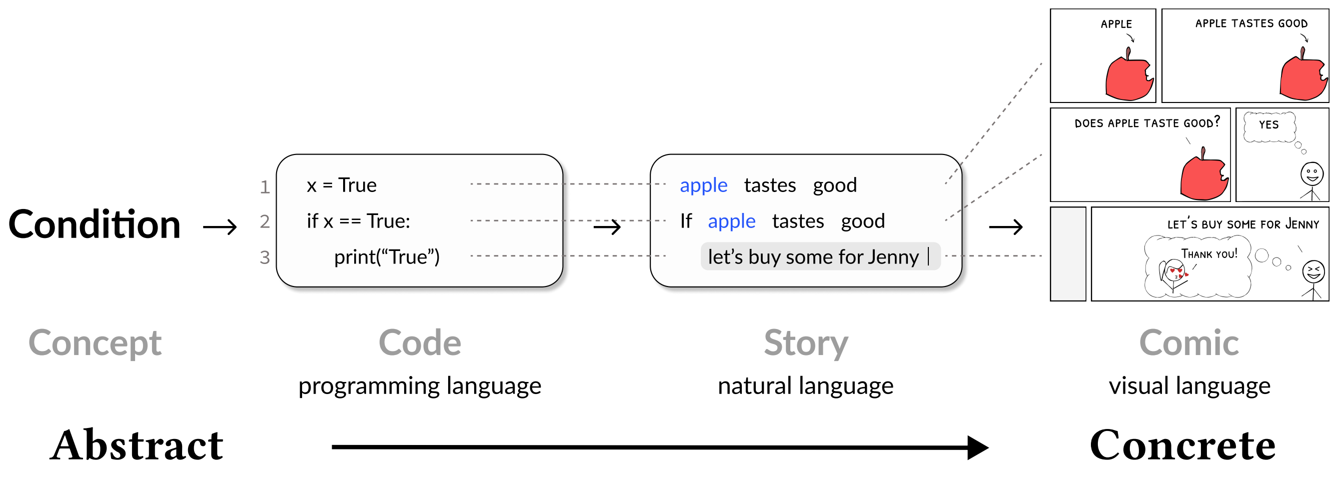
CodeToon: Story Ideation, Auto Comic Generation, and Structure Mapping for Code-Driven Storytelling

Sangho Suh, Jian Zhao, Edith Law
University of Waterloo, Canada
{sangho.suh,jianzhao,edith.law}@uwaterloo.ca



```
1   x = True
2   if x == True:
3       print("True")
```

Figure 1: CodeToon helps users create stories and comics from code. It uses 1-to-1 mapping to make connections clear across code, story, and comic. For example, as indicated by the dotted line, line 1 (code) maps to line 1 (story) and to row 1 (comic).

ABSTRACT

Recent work demonstrated how we can design and use coding strips, a form of comic strips with corresponding code, to enhance teaching and learning in programming. However, creating coding strips is a creative, time-consuming process. Creators have to generate stories from code (code↦story) and design comics from stories (story↦comic). We contribute CodeToon, a comic authoring tool that facilitates this code-driven storytelling process with two mechanisms: (1) story ideation from code using metaphor and (2) automatic comic generation from the story. We conducted a two-part user study that evaluates the tool and the comics generated by participants to test whether CodeToon facilitates the authoring process and helps generate quality comics. Our results show that CodeToon helps users create accurate, informative, and useful coding strips in a significantly shorter time. Overall, this work contributes methods and design guidelines for code-driven storytelling and opens up opportunities for using art to support computer science education.

CCS CONCEPTS

• **Human-centered computing** → **Interactive systems and tools**.

KEYWORDS

code-driven storytelling, comics, coding strip, authoring tool

ACM Reference Format:
Sangho Suh, Jian Zhao, Edith Law. 2022. CodeToon: Story Ideation, Auto Comic Generation, and Structure Mapping for Code-Driven Storytelling. In *The 35th Annual ACM Symposium on User Interface Software and Technology (UIST '22), October 29-November 2, 2022, Bend, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3526113.3545617

1 INTRODUCTION

"Computer science is a field that attracts a different kind of thinker. . . they are individuals who can rapidly change levels of abstraction, simultaneously seeing things 'in the large' and 'in the small'." [19]

Learning programming is difficult due to its abstract nature [38]: it requires learning concepts and programming languages that have been derived through a series of abstractions. Specifically, learning with text-based programming languages poses a barrier for novice learners [20, 32], as text-based programming languages rely on symbolic representations that use a set of seemingly arbitrary rules and abstract expressions. The compact syntax and notations are useful for specifying precise operations but not for communicating to learners the underlying computational ideas intuitively, unlike pictorial or visual representations, which can leverage real-life scenarios to help learners understand computational concepts. As such, novice learners are often forced to memorize the rules and code expressions without understanding the intuitions behind the syntax and semantics. Unfortunately, this has perpetuated an image of computer programming as a set of abstract ideas and rules, and computer science as an abstruse, inaccessible, and uninteresting discipline, especially for those who struggle with abstract reasoning.

To address this, many researches looked at embodied approaches, exploring ways to use familiar abstractions such as real-life objects, situations, and visual representations to make computer programming more concrete and accessible [18, 23, 30, 32]. Recent research on *coding strips*, a form of comic strips with code, follows this line of work by looking at comics as a vehicle. By identifying many design variations and patterns for explaining code executions and semantics, Suh [33] showed that comics can be a powerful medium for visualizing computational concepts and procedures. In another study, Suh et al. [36] tested four use cases of coding strips in an introductory computer science course and found that coding strips can enhance learning in various ways. For instance, one use case included an instructor introducing code expressions with comics first and then with code. Students appreciated this scaffolding over the code-only approach as comics allowed them to learn the intuition without being distracted by the syntax and rules and then pick up code expressions in terms of familiar dialogues and actions.

Unfortunately, despite growing evidence of their usefulness, creating coding strips remains a creative, laborious, and time-consuming process. First, it requires creators to ideate (brainstorm) and select stories that align with code. Second, creators need to invest significant effort and time (and sometimes confidence in drawing) to sketch stories in the form of comics. While Suh et al. [38] proposed a design process and tools to help creators design coding strips, the entire process was manual and not automated [38]. Moreover, while the related literature and previous work suggest that making connections between code and comics obvious is critical for coding strips' success [3, 36], no work has yet explored how we can establish a clear mapping between code and comics.

We introduce CodeToon, a comic authoring tool that supports this creative process with two mechanisms: (1) facilitating—through metaphor recommendation—the ideation of code-aligned stories and (2) automating the generation of comics from stories. Inspired by Gentner's structure mapping theory [16, 17] and visual narrative grammar [12] for comics, the two mechanisms allow CodeToon users to add code or select code examples provided by the tool, generate a story from the code, and automatically produce comics based on the code or story while maintaining 1-to-1 mapping across them. Our two-part evaluation of CodeToon found that this streamlined design process allows users to quickly and easily create quality coding strips that convey a salient connection between code and comics. In summary, our contributions include:

- a computational pipeline that uses story ideation, auto comic generation, and structure mapping for code-driven storytelling;
- CodeToon, a tool for code-driven storytelling where users can efficiently transform code into story, then story into comics;
- user experiments that evaluate the authoring process and the generated results of CodeToon.

2 BACKGROUND

2.1 Building Ladder of Abstraction

Coding strip was inspired by the ladder of abstraction, with comic and code representing different levels within the ladder [1, 32, 33]. Thus, we review previous work that addressed two questions for building the ladder of abstraction: **Q1**: How do we conceive abstractions at different level(s), and **Q2**: What are design considerations?

Q1⇒A: Find what we can abstract over/under. In his interactive article *Up and Down the Ladder of Abstraction* [40], Bret Victor uses variable as a control for moving up and down the abstraction ladder. In this article, the variable is `time`, and a system at a particular `time` an abstraction; readers use a slider to change the `time` (e.g., `t=1` to `t=2`) and observe how the system (abstraction)—a car's trajectory—changes. He equates this interaction as moving up and down the ladder of abstraction, explaining that all systems share the same anatomy—an independent variable (e.g., `time`), structure (the set of rules and what is controlled by the variable), and data (environment)—and suggesting that the process of building the ladder of abstraction (i.e., conceiving abstractions at different levels) consists of identifying what can be parameterized and providing a control to explore the range of abstractions. This informed how we should engineer story ideation (code↦story). Specifically, in conceptualizing how we can turn code into a story, we used this idea to explore what parts of the code can be parameterized and used to develop stories. (Further details in Section 4.)

Q2⇒A: Maintain structure across abstractions. Defining abstraction as "a comparison in which the base domain [(e.g., code)] is an abstract relational structure," Gentner proposed structure mapping theory to posit that the structure—the relations between objects—is the most important factor when conceiving new abstractions, not the number of attributes shared between the base and target domains or the specific content [17]. An example he gives is: "The hydrogen atom is like our solar system." In this example, what makes the hydrogen atom a comparable abstraction is that the hydrogen atom and solar system share the same relation (e.g., the electron REVOLVES around the nucleus, like how the planets REVOLVE around the sun), not their object attributes (they do not share the same object attributes, e.g., color, size, as the planets). A related technique called concreteness fading, which introduces an idea in multiple stages using different representations (abstractions) in decreasing concreteness, also supports this, suggesting that maintaining the relational structure across the representations is the key [14, 31, 37]. This design principle for layers of abstraction inspired us to structure stories and comics to align with the code structure.

2.2 Supporting Comic Authoring

2.2.1 Design Process & Patterns. The time-consuming, laborious nature of creating comics poses critical barriers to their use and adoption. As a result, various approaches have been suggested. For example, to support ideation in the design process, researchers developed design patterns and process with clearly delineated stages to guide authors [6, 38]. Digital authoring tools have been developed to make it easier to quickly draft and iterate on the design by offering templates (e.g., panel layout and images) users can add to the canvas [2, 24, 35]. Our work extends prior work as our tool

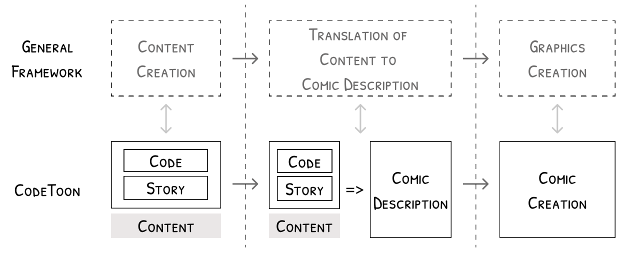


Figure 2: General framework for auto generation of comics suggested by Zeeders [44] and our framework. In our framework, content used to generate comics are code and story. CodeToon users add content (code & story), and the system translates it into comic description and creates a comic strip, the graphic representation of code & story.

supports the authoring of comics (1) based on code input and (2) introduces two new mechanisms—story ideation and auto comic generation—on top of the methods mentioned above.

2.2.2 Automatic Comic Generation. One step forward from supporting a design process with authoring tools, design patterns, and design guidance is automatically generating comics for users [9, 22, 42]. Zeeders [44]—who surveyed auto comic generation methods—suggests that, at a high-level, auto comic generation involves three steps: (1) content creation, (2) translation of content to a comics description, and (3) graphics creation, as shown in Fig. 2. In prior work, the sources of content in the first step have been a multitude of things, e.g., daily activity data [11], chat sessions [26], scripts [29], and movies [21, 43]. (Their work cannot generalize to our content type, code, since it, unlike other data types, lacks contextual information that can be used to form a narrative for a comic without the user intervening to help define a story.) In the second step, they are formatted into a particular format [4, 44] to provide instructions on how they should be presented graphically. The final step is the graphics creation stage, where either the composition or screenshot method is used [44]; the former method composites different images (e.g., character, background, speech bubble) to create a scene for panels, the latter embeds existing scenes (e.g., screenshot of movie scenes) into panels. As we will explain, our work leverages the composition method to generate comics automatically. Overall, our work extends research in this area by demonstrating how we can auto generate comics from a new content type, code.

3 CODETOON

3.1 Design Goals

To develop CodeToon, we first conducted a pilot study with 12 participants. Two participants (age: M=44.5; gender: 1F, 1M) were teachers with 5+ years of experience teaching programming; ten other participants (age: M=27.9; gender: 5F, 5M) were undergraduate and graduate students with varying teaching experience (6 0-1 year, 2 1-3 years, 1 5+ years). We chose teacher and student participants highly experienced in programming (11 Much Experience, 1 Some Experience) as opposed to participants without programming experience in order to harness the insights they picked up over the years as teachers and students. Over the course of the pilot sessions, we improved, added, and tested new features and workflows of CodeToon (Sections 3.2 and 4) until we could observe that no major changes are needed to enable a creative authoring experience. Based on this pilot study, the literature on multiple representational systems [3, 8, 37], and creativity support for comics [38], we derived the following design goals for CodeToon:

D1. Allow users to iterate on their code, story, and comic. From our pilot study, we found that the authoring process may not be linear. While creating comics, users can be inspired by their comics and form additional ideas to add to their story. While working on the story template, they could also think of a better story and desire to edit code (e.g., change the value assigned to a variable). Thus, the tool should make this interaction easy for its users.

D2. Augment, not constrain, users' creativity with our story ideation and auto comic generation. Our pilot study revealed that providing story ideas and comic templates can accelerate the authoring process. But it also showed that some users can already have some ideas on what they want to create and how to design their comics. Thus, the tool should not limit users to using only story ideas and comic templates provided by the tool.

D3. Make mapping clear across code, story, and comic. Research suggests that making the correspondence explicit and consistent is essential when presenting multiple representations [36, 37]. Otherwise, they do more harm than good because they only confuse people. For us, this means that the mapping (↔) between code, story, and comic should be clear. Previous research on coding strip also found that the mapping between code and comics needs to be clear for it to be effective and useful [36].

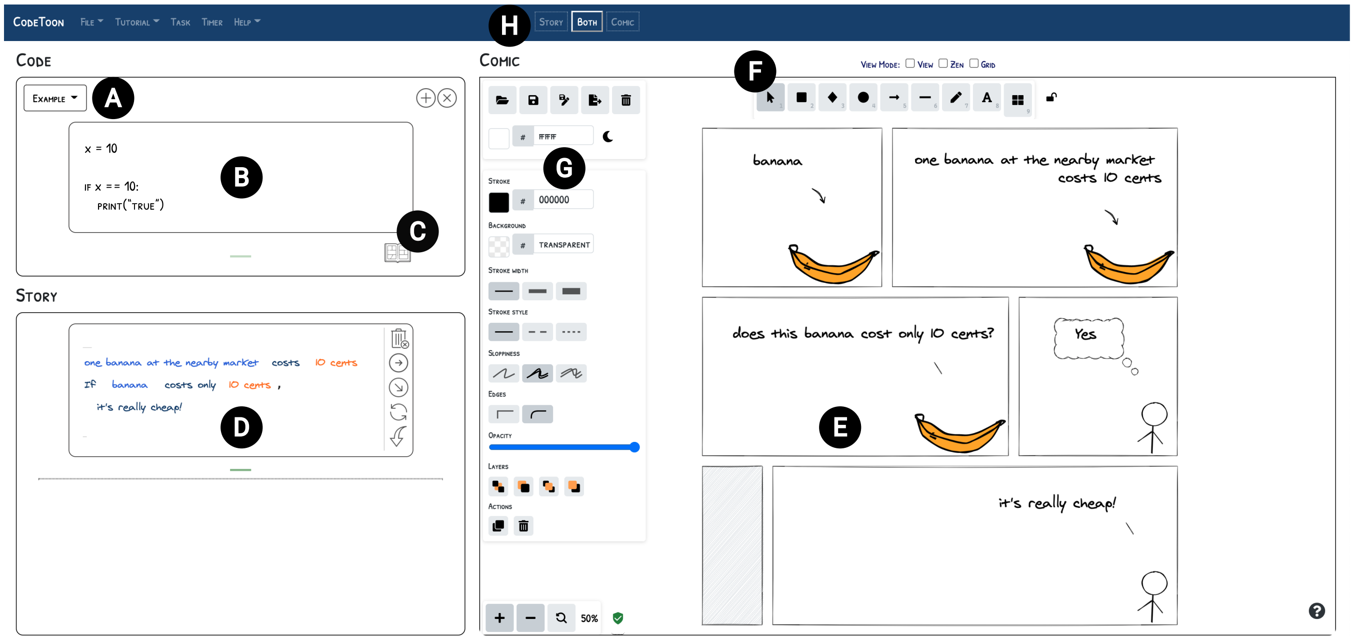
D4. Use simple, scalable visual vocabulary. Scalability relies on having a set of basic building blocks that can be combined to build anything of varying complexity (cf. the composition method in Section 2.2.2). A set of building blocks in visualizations is called visual vocabulary. To generate comics that can scale to any code input, establishing a simple, scalable set of comic templates that can be easily combined to express any set of code is necessary.

As a whole, our design goals aimed to create a 'low floor, high ceiling' system for generating coding strips. That is, a system that makes the process of creating coding strips simple, effortless, and easy, while providing a high ceiling for creative exploration.

3.2 User Interface

CodeToon consists of three panels: code (Fig. 3B), story (Fig. 3D), and comic (Fig. 3E). Users can select any layout button (Fig. 3H) to change which panels are shown. The default layout (Both) shows all three panels (Fig. 3), the Story layout code and story panels, and the Comic layout the drawing canvas. The layout feature allows users to customize the workspace—the number and layout of the panels—for an optimal experience.

The three panels represent stages in the design process for creating a coding strip [38]. Concretely, the basic workflow consists of users (1) adding code in the code panel (Fig. 3B), (2) generating a story template (from the code) and writing a story in the story panel (Fig. 3D), and (3) using the auto comic generation feature to instantly generate comic in the comic panel (Fig. 3E). Below, we describe each panel and how they facilitate this workflow.



```
x = 10

if x == 10:
    print("true")
```

one banana at the nearby market costs 10 cents
If banana costs only 10 cents ,
it's really cheap!

banana
one banana at the nearby market costs 10 cents
does this banana cost only 10 cents?
Yes
it's really cheap!

Figure 3: System interface: (A) drop-down allows users to check potential code examples for basic programming concepts, (B) code container, (C) button that generates story template from code in code container, (D) story template, (E) drawing canvas for comics, (F) tool palette, (G) style palette, and (H) buttons for changing the interface layout (between code & story, current, or canvas-only layout).

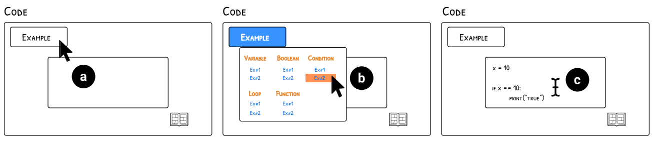


Figure 4: Users can add code by selecting code example (b) or by typing (c).

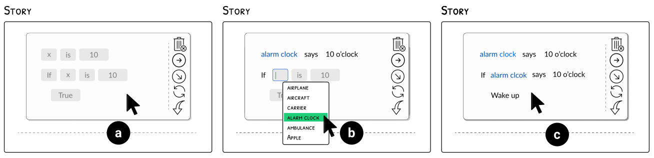


Figure 5: Users can add story (to story template) by selecting a list of ideas in dropdown (b) or by typing into input box.

3.2.1 Code. In the code panel (Fig. 3B), users can add any number of programs, each within a different code container. As shown in Fig. 4, users can add the code to the container by using the code example repository (Fig. 3A) or manually typing into the code container. There are two buttons in the top right corner of the code panel for adding (⊕) and deleting (⊗) code containers. The ability to add additional code containers was added during the pilot phase to make it easy for users to iterate on their code, story, and comic (**D1**). After the user adds code, they can press a button (Fig. 3C,) to generate a story template (Fig. 3D) from code.

3.2.2 Story. When a user generates a story template, it is added to the story panel, which is initially an empty panel. Fig. 3D shows what the user would see when a story template is added. Story templates are linguistic representation of the code, with input boxes where users can add real-life equivalents for the code expressions. For instance, the code expression `x = 10` generates `x` `is` `10` (Fig. 5(a-b)) as its story template. Now, a user can add `alarm clock` to `x`, `says` to `is`, and `10 o'clock` to `10`. As shown in Fig. 5(b), CodeToon provides a dropdown containing a list of metaphors to help users brainstorm story ideas. The dropdown does not appear for every input box, however. At the time of testing, it appeared only on input boxes mapped to variables (e.g., x) and assignment operators (=). The dropdown for the former showed a list of 345 categories (e.g., apple, car) and that of verbs synonymous with or semantically close or related with the semantics of assignment operator (e.g., assign, has) for the latter. While dropdowns appear to help users with story ideation, they do not have to form their stories around these suggestions; they can type any text into the input box to create any story (**D2**).

3.2.3 Comic. As shown in Fig. 6, a user can instantly generate a comic (Fig. 3E) by selecting any of the two arrow icons (→ and ↘) below the trash can icon . The reason for the two arrows is to offer users the flexibility to expand to the right or below the existing drawings. If users change the story, they can press the update icon ↻ (below the arrow icons) to instantly update the content of the auto generated comic to reflect these changes, making iterative design of their story and comic frictionless (**D1**). While users can use the auto comic generation feature to instantly generate comics

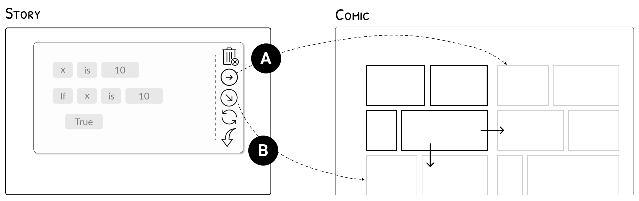


Figure 6: Users can generate a comic by selecting either of the arrow buttons. If canvas already has some drawing elements, as in this case, they can add the comic to the right (A) or below (B) the existing elements. If canvas is empty, both buttons place the comic at the center of the canvas.

from the story template, they can also use the tool palette (Fig. 3F), style palette (Fig. 3G), and library (Fig. 7) to manually create or edit/expand on the auto generated comic. The view mode above the palette (Fig. 3F) provides three checkboxes for turning on/off different view modes: Grid makes the grid appear in the canvas for precise alignment and measurement, Zen removes style palette, and View removes both the tool and style palette and makes the canvas view only.

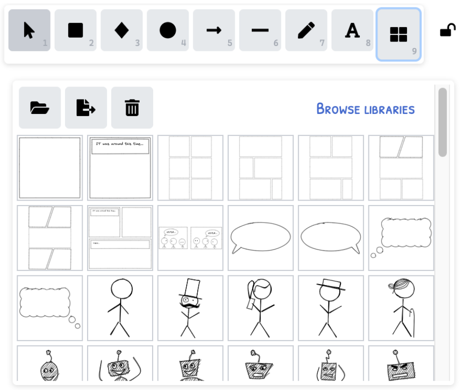
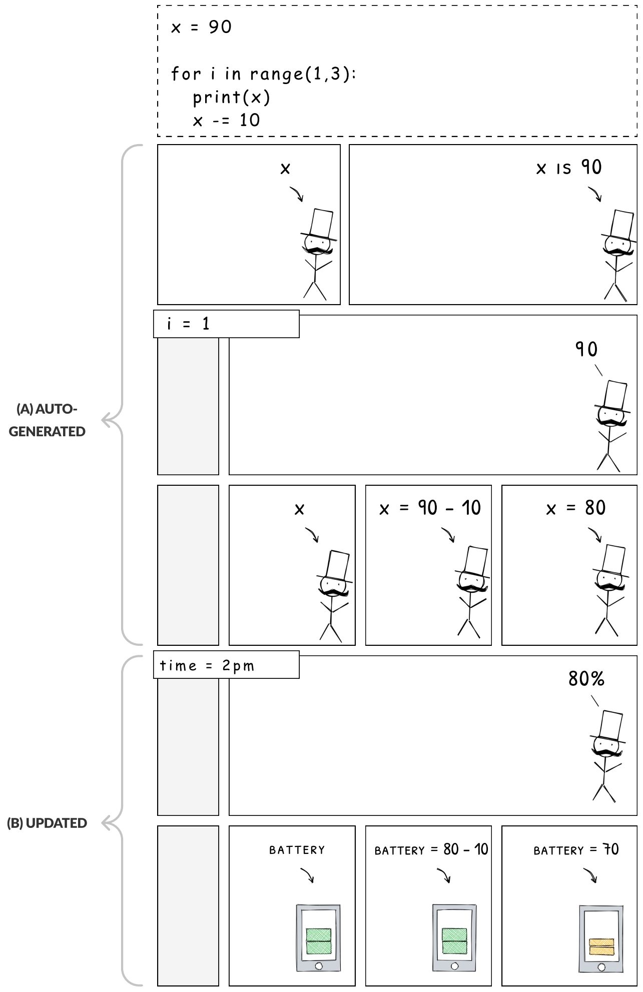


Figure 7: The library offers pre-drawn templates, such as panels, speech bubbles, and characters, for creating comics.

3.3 Usage Scenarios

To further clarify the user interface and workflow, we present two scenarios to demonstrate how users can use CodeToon.

Teacher. Amanda is a high school teacher. She is teaching a programming class to 10th graders who are learning programming for the first time. She wants her students to discover that programming is not just about memorizing rules, syntax, and expressions. She wants her students to realize that computational ideas can also be found in our daily lives. When the class starts, she explains this as her goal for her students. To show how they can think of computing in terms of real-life objects and situations, she opens CodeToon and shows a sequence of the code-story-comic example shown in Fig. 1. She explains that while code expressions may appear scary for now, they can think of code as being no different from words we use to communicate. To direct their attention to the logic (i.e., if-then structure) and not the content, she switches the object in the story from apple to banana and updates the comic (by pressing the update icon). To make the stories more engaging and help students connect with them, she also adds contextual details to the expression banana, editing it to one banana at the nearby market, generating a sentence one banana at the nearby market costs 10 cents (cf. Fig. 3). She invites her students to suggest a story and produces comics for them on the fly. Students suggest diverse stories based on their experience and learn that programming is not as difficult as they thought it would be.


```
x = 90

for i in range(1,3):
   print(x)
   x -= 10
```

Figure 8: A simple loop code and (A) auto generated comic. The bottom two rows represent (B) comic updated with different images (phone) and text (e.g., BATTERY for x). Like in this example, CodeToon can update specific rows, giving users fine-grained control over their stories.

Student. Jane is a graduate student working as a teaching assistant in the introductory CS course. While overseeing a lab, a student asks for help, saying his code is not working the way he wants it to. She takes a look at the code and feels that a piece of code involving loop may be the culprit. She tells the student but realizes that he does not have a good grip on the concept of loop and how to debug the code. She opens CodeToon and adds the code in his assignment and generates a comic to show the student how loop works. Fig. 8 shows an example of a code with loop and its corresponding comic. Jane explains to the student how the first row of the comic maps to the first line of the code (`x = 90`) and that the next two rows represents the two lines in the loop; She notes that the program then moves to the next iteration and points to `i = 1` as an indicator of which iteration the program is running at. To help the student connect the concept to a real-life situation, Jane adds a story to the story template: she replaces `x` with `my phone's battery`, `=` with `is at`, `90` with `90 percent`, and `i` with `time` and presses the update icon to update the comic, which shows a comic with a story that shows the phone having a battery initially at 90% and decreasing by 10% every hour, as shown in Fig. 8. Jane creates another code container and shows the student another loop example to help him master the concept. After the student is done receiving help from Jane, he asks Jane for the URL of CodeToon so that he can use it to review and assist his studies in the class.

4 CODE-DRIVEN STORYTELLING

CodeToon supports code-driven storytelling by generating comics from code using two mechanisms—story ideation and auto comic generation. In this section, we describe how we designed them. The implementation details of the computational pipeline are described in Fig. 18 in Appendix.

4.1 Story Ideation

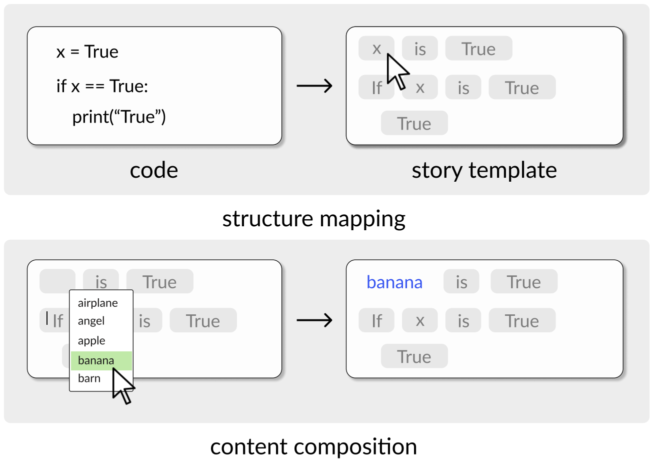


Figure 9: Our story ideation consists of generating a story template aligned with the code structure (structure mapping) and allowing users to fill the template with the help of metaphor suggestions (content composition).

CodeToon aims to make the generation of comics from code more efficient, and the first step in that process is rapid story ideation. To understand how we can turn code into a story, we first went through code expressions and turned them into a story. Table 1 shows some examples. We found that any code expression can be parameterized and replaced with metaphors of corresponding form (cf. Section 2.1). For instance, variables (e.g., `x`) can be metaphors in noun (e.g., wallet) or phrases (e.g., message in my email); assignment operator (=) can be *be* verbs (e.g., am, is, are) and *transitive & intransitive* verbs (e.g., wallet *has* 5 parking coins, my dog *feels* sick); values can be contextualized (e.g., 5⟶5 o'clock, `True/False`⟶on/off, `''hello''`⟶"hello, John"); keywords (e.g., `def`) and (built-in/user-defined) function names (e.g., `print()`) can be replaced with semantically related verbs/phrases. For instance, `print` can be say. Therefore, in the story template, we (1) provided a list of metaphors they can choose from and (2) converted code expressions into text fields, to allow users to be creative with the story authoring (**D2**).

Table 1: Examples of code and corresponding stories

code	*hybrid* (code & story)	story
`x = 5`	*time = 5* *wallet = 5* *student = 5*	time is 5 o'clock wallet has 5 parking coins student received 5 dollars
`x = True`	*switch = on* *my_schedule = busy* *this = True*	switch is on my schedule is busy this is expensive
`x = ''hello''`	*message = "hello"*	message reads, "hello"
`print(''Even'')`	*print("it's even")*	say, "It's even!"

4.2 Auto Comic Generation

Auto comic generation requires the conversion of different types of code expressions into a visual panel arrangement (e.g., panels, characters, speech bubbles). We approach this problem by defining in advance a specific design template for each code expression (e.g., variable assignment, loop). Specifically, we leveraged the theory of Visual Narrative Grammar (VNG) [13], which suggests that each panel of a comic can be categorized into one of the five phases of a narrative: (1) Establisher, which sets up a scene; (2) Initial, which depicts the start of an action; (3) Prolongation, which shows moments between the start of an action and its peak; (4) Peak, which marks the point in the action when the tension reaches the peak; (5) Release, which shows moments after the action has ended. Below, we provide two examples of code expression template, and the thought process that went into designing these templates using VNG.

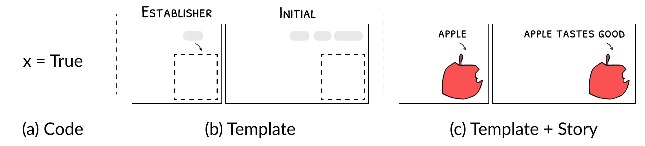


Figure 10: Variable assignment: code, template, example

Fig. 10 shows the variable assignment `x = True` (Fig. 10(a)), its template (Fig. 10(b)), and the first row of the comic in Fig. 1 (Fig. 10(c)) that uses this template. Variable assignment can be defined by two operations: first, computers allocate a memory space for a variable; then they assign value to the variable. The first operation is (semantically speaking) analogous to Establisher in that it *sets up a scene* (for assigning value). The second step can be considered as Initial as it *initiates* the action of assigning value to this variable. Hence, we use two panels, Establisher and Initial, as shown in Fig. 10. While it can also be valid to have just one panel (e.g., Initial instead of Establisher + Initial), one consideration that led us to choose the Establisher + Initial combination is design. Typical computer programs will contain multiple variable assignments, which means that the auto generated comic will have several templates like this, stacked on top of each other. Having two or more panels in each row, aligned and stacked on top of each other, can make the final comic design look more structured (cf. Fig. 1) than the design where multiple rows have only one panel in each row. Additionally, this design can be more useful in the classroom. When teaching variable assignment, for example, having illustrations for space allocation and value assignment can allow teachers to teach what happens at the memory level.

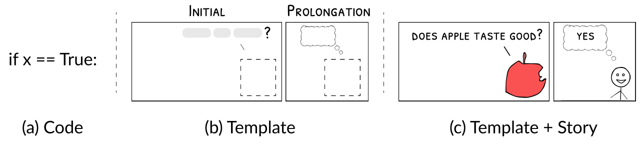


Figure 11: Conditional expression: code, template, example

As another example, Fig. 11 shows the conditional expression `if x == True:` (Fig. 11(a)), its template (Fig. 11(b)), and the second row of the comic in Fig. 1 (Fig. 10(c)) that uses this template. A similar thought process went into designing this template: the conditional expression first checks whether the statement is `True` or `False`; as this *initiates* an action, the first panel is an Initial panel and has a placeholder for text that ends with a question mark to indicate the checking action. The following panel is used to report (hence the speech bubble) whether the expression evaluates to `True` or `False`; this panel is a Prolongation panel, as it sits between Initial and Peak (code wrapped around the conditional expression that would run if the expression evaluates to `True`).

There can be more than one way to define these templates for coding strips depending on the goal(s), programming language and paradigm. In our case, the goals were: ensuring scalability, conveying semantics and executions, and offering design variations (e.g., if possible, avoid using only a single panel within a row); also, our visual vocabulary was created using Python (code expressions, syntax, and conventions). Understanding various nuances required to construct visual vocabularies for various programming languages is not within the scope of this research; we leave that as future work.

A key challenge in generating comics from code is maintaining a clear mapping between code, story, and comic (**D3**), so that learners can tell which line of code maps to which line of the story and which panel of the comic. We achieved this design goal (**D3**) by generating story template and comics that map 1-to-1 to lines of the code, as shown in Figs.1 and 9. In addition to the 1-to-1 mapping, visual cues were also used to make the correspondence easily perceptible. For example, if there was an indentation in the code, this was carried over to the story template. For comics, an empty gray panel was

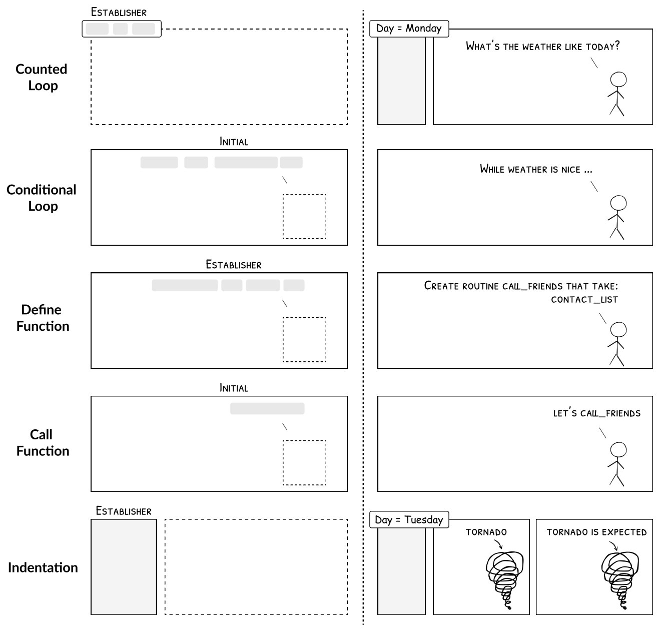


Figure 12: Other comic templates (left) and examples (right): the gray panels represent indentation; the dotted panels comic templates (e.g., variable assignment, counted loop); the gray boxes inside panels text (e.g., Day = Monday, while weather is nice); the dotted squares inside panels objects (e.g., stick figure, tornado, apple, banana).

used to indicate indentation, as shown in Fig. 12. Note that, as shown in Fig. 8, when code has loop, the 1-to-1 mapping cannot be maintained as the comic needs to visualize the iterations.

5 EVALUATION

To test whether CodeToon successfully supports the creative design process and generates quality comics from code, we conducted a two-part evaluation: a user study and a comic evaluation survey.

In the user study, we aimed to answer: **(RQ1)** Does CodeToon support the authoring of coding strip, in terms of story ideation and comic creation; **(RQ2)** Does CodeToon make the process of authoring coding strip more efficient; and **(RQ3)** What are the perceived utility and use cases of CodeToon for teaching and learning programming? Note that **RQ3** does not focus on how well CodeToon supports the authoring of coding strip but on the perceived pedagogical value of CodeToon. While recent studies [36, 38] revealed learning benefits in participating in the process of creating coding strips and learning with coding strips, these studies were not done with CodeToon. This motivated us to explore **RQ3**, as it can reveal potential benefits and guide future work. Further, based on the comics created from the user study, in our comic evaluation study, we aimed to investigate: **(RQ4)** Does CodeToon help generate high-quality comics, and how consistent is the quality?

5.1 Part 1: User Study (RQ1–RQ3)

Our user study employed a between-subjects design with two conditions: Baseline (B) and CodeToon (C). Baseline was the same as CodeToon, but without the story ideation and auto comic generation features. Baseline users could generate story template from code. However, they did not have access to metaphor suggestions (Fig. 9) and buttons (Fig. 6) to instantly generate comics from the story template. Baseline users had to brainstorm story ideas on their own and manually create comics using the templates from the library (Fig. 7) and drawing tools (Fig. 3 (F&G)).

5.1.1 Participants. We recruited all 24 participants (12 for each condition) from a local R1 university's study participant recruitment platform. Participants were required to have (1) basic programming knowledge, (2) a mouse, and (3) Chrome browser on their device.

Most Baseline users (age: M=23.3, SD=2.5; gender: 8F, 4M) had decent programming experience (6 Some Experience, 6 Much Experience), some experience with digital drawing tools (2 No Experience, 9 Some Experience, 1 Much Experience), and mostly positive perception towards the comics' usefulness as a tool for teaching and learning programming (1 Slightly, 1 Moderately, 6 Very, 4 Extremely Useful). Many were in the middle in terms of their confidence in drawing (4 Not Confident, 5 Somewhat confident, 3 Confident) and creating comics for teaching programming concepts (3 Not Confident, 8 Somewhat Confident, 1 Confident). They had varied experience with teaching programming (6 No Experience, 2 0-1 year, 4 1-3 years).

CodeToon users (age: M=27.3, SD=4.8; gender: 2F, 10M) also had decent programming experience (9 Some Experience, 3 Much Experience), some experience with digital drawing tools (3 No Experience, 7 Some Experience, 2 Much Experience), and similar perception towards the comics' usefulness (1 Slightly, 2 Moderately, 4 Very, 5 Extremely Useful). Many of them were not confident in drawing (7 Not Confident, 4 Somewhat Confident, 1 Confident) and creating comics for teaching programming concepts (8 Not Confident, 3 Somewhat Confident, 1 Confident). They had little to some experience teaching programming (3 No Experience, 5 0-1 year, 2 1-3 years, 1 3-5 years, 1 5+ years).

5.1.2 Procedure. Baseline and CodeToon users followed the same study procedure. The study was conducted remotely using video conference software. Participants first completed a pre-study survey. The survey included questions about their demographic information. Then, they went through tutorial videos that explained the interface and how to use the tool, after which participants conducted practice tasks—replication tasks by following the videos. CodeToon users spent more time (at least 8 min) in the tutorial phase as they needed to learn and try story ideation and auto comic generation features.

Next, participants entered the task phase. The task was to create a comic about a programming concept of their choice. They were instructed that the end goal is to have code and corresponding comic that can be used together to teach students about the concept. Time limit was not imposed to study how users perform the task in a natural setting. Through pilot studies, we knew that participants generally had enough time to complete the task.

After the task phase, we administered three surveys: (1) post-study survey with creativity support index (CSI) [10], (2) paired factor comparison (PFC) [10], and (3) system usability survey (SUS) [7]. We also conducted a short interview to ask participants to elaborate on their survey response to get a better understanding of what worked and what did not work. The study lasted between 1.5-2 hours and the participants received a $30 Amazon gift card for their participation.

5.2 Part 2: Comic Evaluation Study (RQ4)

Our comic evaluation study followed the same procedure as in prior work [28], comparing comics resulting from two different conditions (Baseline and CodeToon) to understand whether CodeToon helped generate comics of better quality.

5.2.1 Study Dataset. After our user study, our comic dataset[1] consisted of 24 comics covering VARIABLE (1B, 0C), CONDITION (7B, 7C), LOOP (2B, 2C), and FUNCTION (2B, 3C). We selected a subset instead of all comics for our survey for two reasons. First, it was not realistic to ask participants to rate all coding strips, as that can take approximately 2 hours (if 5 minutes per comic). We did not want to risk a survey receiving poor responses due to its length [15, 25]. Second, the quality of Baseline comics varied greatly. Since the amount of effort participants exerted ranged from very little to very high, we chose to select quality comics that participants put effort into (e.g., those who indicated they were satisfied with their work). In this way, we could minimize variability and keep the survey at a reasonable length.

Thus, we formed pair of comics (Baseline vs CodeToon) for the concepts by filtering for comics where participants answered in the post-study survey that they were 'highly satisfied' with their comics (9 or 10 out of 10 on 'I was satisfied with what I got out of the tool'), because participants' level of satisfaction varied (B: M=8.4, SD=1.6, Range=[6,10]; C: M=8.9, SD=1.3, Range=[6,10]). Then we grouped them into sets based on their concepts. This resulted in 2 sets for CONDITION, 1 set for LOOP, and 1 set for FUNCTION, in total 8 comics to rate for a 30 to 55-minute survey.

5.2.2 Participants. We recruited 20 participants (age: M=24, SD=3.9) who can read and understand basic Python code through R1 university's study participant recruitment platform. Participants were mostly proficient in coding (1 Beginner, 1 Semi-Amateur, 4 Amateur, 8 Semi-pro, 6 Pro) and had moderate attitude towards comics' usefulness as a tool for teaching (2 Not At All, 3 Slightly, 8 Moderately, 4 Very, 1 Extremely Useful) and learning (3 Not At All, 1 Slightly, 9 Moderately, 4 Very, 2 Extremely Useful).

5.2.3 Procedure. After signing up to participate, participants received the Qualtrics survey URL. After demographic questions, the survey presented eight comics in random order. Each page started with the programming concept the comic is based on, the comic, the code the comic is based on, and then a set of evaluation questions related to (1) how well the comic maps to code, (2) how well the comic illustrates code and concept, and (3) how useful they think the comic is for teaching and learning the concept it is based on. Participants received $10 Amazon gift card for their participation.

[1]Downloadable at https://codetoon-research.github.io/download/

6 RESULTS

Baseline comics were generally inconsistent in their design language compared to CodeToon comics, as Baseline users manually designed their comics while CodeToon users incorporated auto generated comics. Thus, CodeToon comics were generally longer and more structured than Baseline comics. As for usability, both the Baseline and CodeToon users found the tool highly usable (SUS_B: 78.5, SD=20.5; SUS_C: 81.0, SD=9.6), giving usability scores well past the cutoff score (68) for production. For anonymity, we use B1…B12 and C1…C12 to refer to Baseline and CodeToon users, respectively.

6.1 RQ1: Does CodeToon support the authoring of coding strips?

We analyze a mix of responses from the survey, interview, and CSI results. We review the effectiveness of the proposed two features—story ideation and auto comic generation—as well as whether participants found the code-to-comic mapping accurate and the tool helped them be creative. We begin with Fig. 13, which summarizes the participants' ratings on the usefulness of the two novel features.

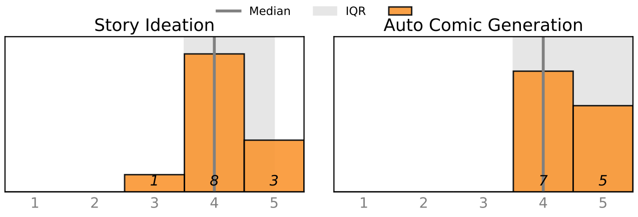


Figure 13: CodeToon (1: Not At All; 5: Extremely Useful)

6.1.1 *Feature 1. Story Ideation.*

CodeToon users liked how story ideation helped "spark creativity" (C5) and "think of creative ideas" (C3). C6 noted that "the options [in the dropdown]" provided "initial push of ideas" to get started on figuring out what kinds of "objects or things" could be thought of. C8 was quite surprised that seeing the list of ideas could inspire him to create "much more content beyond this object." C2 appreciated the fact that even though they had "the extensive list of the objects that [they] could choose" from, they could also write anything inside the input boxes (suggesting CodeToon satisfied one of our design goals, **D2**).

6.1.2 *Feature 2. Auto Comic Generation.*

CodeToon users liked the auto comic generation feature because it saves "a lot of time creating the layout" (C12). C3 complimented CodeToon as a system that—even compared to other commercial comic drawing tools—is more useful, because it "automatically generates the comics," decreasing the "workload by a big factor." Several participants identified the auto comic generation as a novel feature, asking whether there has been work like this before.

6.1.3 *Impact of Two Features.*

Fig. 14 provides further evidence that the two features facilitated the authoring of coding strips. Through the comparison, where the difference between CodeToon and Baseline is the presence of the two features, we see that they increased the perceived usefulness of the tool and decreased the perceived difficulty of the task.

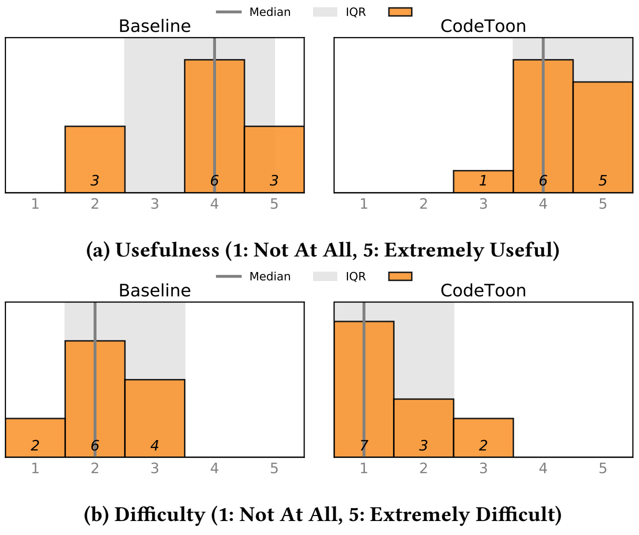


Figure 14: Baseline vs CodeToon comparison on the (1) usefulness of the tool for the task and (2) the difficulty of the task with the tool. CodeToon users found the tool more useful for creating comics from code and the task less difficult.

6.1.4 *Code-to-Comic Mapping Accuracy.*

To understand whether CodeToon users think that the generated comics illustrate code executions and semantics accurately, we asked them in the survey. As shown in Fig. 15, they mostly found the generated comics' overall accuracy and their accuracy with the code executions and semantics very accurate. Many participants supported our design decision to map each line of the code to each row of the comic. C5 explained that she gave a high score to these accuracy questions because the generated comics mapped to the code line by line. C3 explained that this is why, compared to before the task, he feels comics are more useful. C3 said, "I never thought [you] can teach the if conditions and conditional loops and everything to a student in the form of comics [in this manner]." C5, who prior to using CodeToon thought about an alternative "event-by-event approach," acknowledged that this design choice is better, saying: "…it is very helpful to have comics that go along with every single line just so that they can see what each line is actually doing. It is kind of like walking through the code step by step in a visual way…I think it is better that it is line by line, because [it] makes it a lot easier for learners to make that connection." C8 made an interesting remark that accurate mapping between the comic and code can help address concerns about metaphors sometimes failing to communicate the ideas accurately and that if one were to use comics to teach programming, it has to be mapped to code like in CodeToon:

> "I find that this tool maps the comic to the code very well…more accurate than I have ever expected…this balances the tradeoff between the barrier of learning the code and the inaccuracy of the metaphor. If teachers are to tell the student that the [comic] frames map to certain parts of the code, it has to be like this. Strictly mapped to the structure [of the code]…this changed my thoughts on how helpful comic is." (C8)

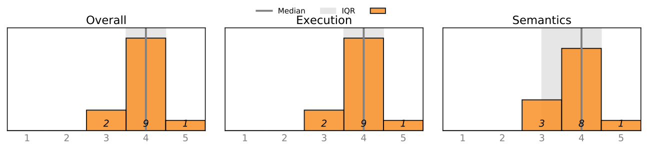


Figure 15: Mapping accuracy for auto generated comics. (1: Not At All, 5: Extremely Accurate)

6.1.5 Creativity Support Index. As shown in Table 2, CodeToon scored high and better than Baseline in all factors of the Creativity Support Index. There was a statistically significant difference in enjoyment, suggesting participants highly enjoyed using CodeToon. In fact, during the interview, many CodeToon users mentioned how fun the tool was. C1 said, "This is a really interesting work. I really enjoyed playing with it." Moreover, all the CodeToon users except for one (11 More Confident, 1 Same as Before) who was already 'Somewhat Confident' in creating comics about programming concepts before the task answered that they feel 'More Confident' about creating comics about programming concepts after this task.

Table 2: Creativity Support Index Results. CodeToon performed better on every factor in creativity support. Statistical significance (p < 0.05) is marked with *.

	Baseline	CodeToon	*Sig.*
Factor	Score (SD)	Score (SD)	p
Enjoyment*	15.9 (1.56)	**17.6** (2.0)	**0.01**
Expressiveness	16 (3.1)	**16.8** (1.5)	0.21
Exploration	16 (3.1)	**16.4** (1.6)	0.39
Immersion	15.8 (4.1)	**16.8** (2.6)	0.24
Results Worth Effort	16.6 (2.4)	**17.8** (2.1)	0.10
Overall CSI Score	80 (14.6)	**85.2** (7.5)	0.14

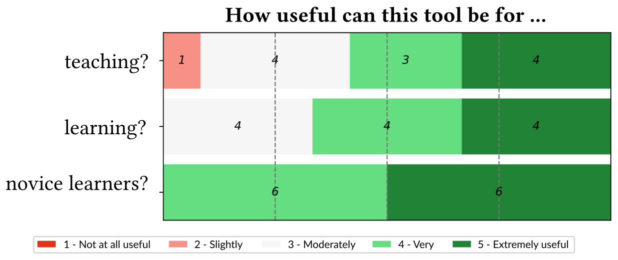
6.2 RQ2: Does CodeToon make the process of authoring coding strips more efficient?

Our analysis showed that CodeToon users were able to save more than 6 minutes on average for the comic authoring (T-test: p=0.08; Baseline: M=18:35, SD=11:17; CodeToon: M=11:45, SD=6:20) and the overall authoring time (Baseline: M=24:47, SD=13:43; CodeToon: M=18:27, SD=9:16). The average time spent on creating a story, on the other hand, was almost the same (T-test: p=0.7; Baseline: M=6:11, SD=3:13; CodeToon: M=6:41, SD=3:38), which can be attributed to the relatively less time-consuming and less challenging nature of creating stories when compared to creating a comic. While CodeToon saved more than 6 minutes, the overall time difference was statistically insignificant (T-test: p=0.19).

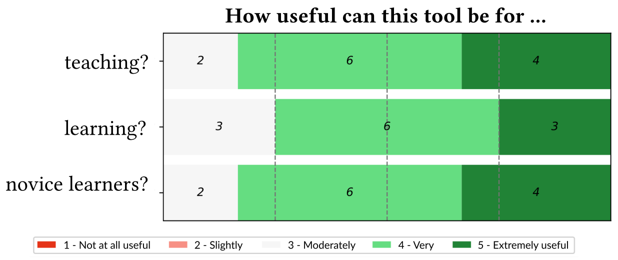
That CodeToon users spent less time is not surprising, given that CodeToon can instantly produce comics. After CodeToon users generated comics, they either submitted them as they are or spent some time simply adding a few details, e.g., additional panels before or after the generated comics to add additional context to the story. CodeToon users generally seemed satisfied with the layout of the generated comics. Although they were told they could edit them, none of them did. C3, who uses a PowerPoint in the workplace to explain code flow to coworkers, was impressed with how well CodeToon automatically generates a nice visualization to illustrate the code flow, saying "[after CodeToon] automatically generates [comics,] the only thing [left to do] is the finishing touch."

6.3 RQ3: What are the perceived utility and use cases of CodeToon for teaching and learning programming?

6.3.1 Perceived Utility. As shown in Fig. 16, most CodeToon users perceived CodeToon as a very and extremely useful tool for teaching, learning, and novice learners. Most of them (8/12) said they would like to use the tool for teaching; the others (4/12) who answered 'Maybe' explained that this is because whether CodeToon is the right tool can depend on "age," "programming level," or "learning styles."[2] Likewise, there were variations in the perceived utility of CodeToon. While several participants mentioned that the tool would be especially useful for "younger students" (C9), others suggested that it can be useful for older students (C8), such as undergraduate students (C5), and even for experienced programmers, e.g,. for "debugging" (C12). One thing everyone seemed to agree was that CodeToon provides a fun, creative way to learn programming.



(a) Baseline



(b) CodeToon

Figure 16: Perceived utility of Baseline and CodeToon for teaching and learning programming

Participants provided several reasons for positively rating its utility. They thought that CodeToon can help students master "important programming concepts," namely loop and function (C12). C12 said, "especially in loop where [automatically generated comics] show `i=0`, `i=1`, ...comics give a visual explanation very well." C2

[2]While we do not acknowledge the notion of "learning styles," it has been added to avoid any loss of information or connotation.

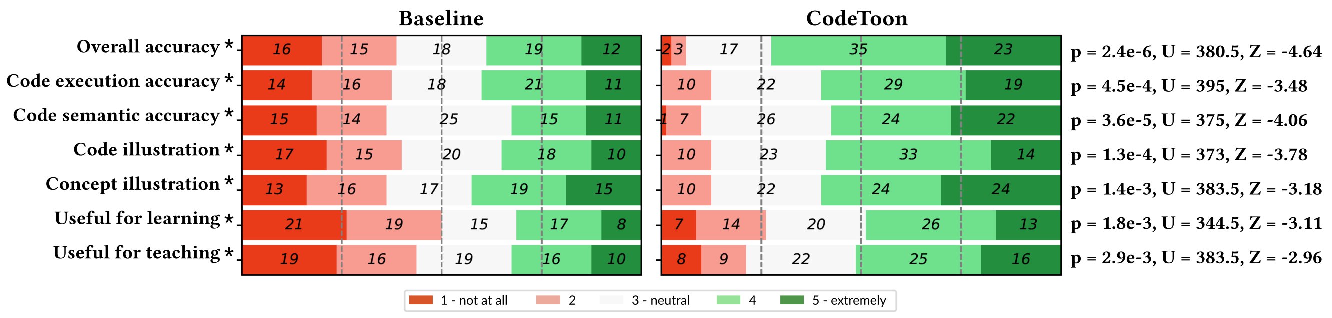


Figure 17: Comic evaluation results. Statistical significance (p <0.05) is marked with *.

suggested that CodeToon would be useful for teaching data types to beginner students as it can show various examples (e.g., INTEGER: apple costs 10 dollars; BOOLEAN: apple tastes good) to help develop intuitive understanding. C6 and C10 stated that the "scaffolding" in CodeToon would help students develop computational thinking, enabling them to see computational ideas in real-life situations and vice versa (i.e., moving between different abstraction levels, as shown in Fig. 1). C6 noted that learning with CodeToon can show how "computer science can be engaged beyond looking at codes on a screen. . . [and that comic] is a very good lens to use."

Most CodeToon participants (9 More Useful, 3 Same as Before) stated that the experience of using CodeToon made them realize that teaching/learning with comics (created from this tool) can be more useful than they had thought. Several participants explained that they had no idea that comics could be used in such a meaningful way. They explained that they thought comics were going to be used to merely narrate programming concepts. But seeing CodeToon generate comics that map to code line by line made them realize that comics can be "very useful" (C3).

6.3.2 Use Cases. Participants provided several ways the tool can be used for teaching and learning programming. One group of related ideas was using the task as an exercise in "classroom" or "tutoring settings" as well as a group or individual assignment, after which they can "present to class [their] story and understanding of the code." Another idea was holding a "special programming class" where students learn to read code after learning their corresponding comic expressions and then using it for exams or quiz where students—instead of explaining what a piece of code means—"can make comics to show what they mean." Participants also made several suggestions pertaining to how to use the tool. B6 suggested "start[ing] with the comic first and then go[ing] to the story [and then] to the actual code," saying that "this would help generalize the topic very easily." Several participants suggested using it to "visualize" basic programming concepts including "data types" as well as complicated concepts such as "pointer" (B12). Explaining how teachers utilize funny comic strips or memes in their teaching slides, participants also suggested that teachers can use the tool to quickly create and add visuals (comic) to their slides. C3 noted that the story ideation feature can also help teachers come up with appropriate examples and metaphors to explain the code and programming concept.

6.4 RQ4: Does CodeToon help generate high-quality comics?

Fig. 17 shows how Baseline and CodeToon comics compare across the measures we investigated. As shown, there were statistically significant differences (Exact Wilcoxon-Mann-Whitney Test) between Baseline and CodeToon comics across all measures (accuracy, illustration, usefulness). In all measures, CodeToon comics were rated better than Baseline comics. That is, CodeToon comics were perceived as more accurate, illustrative, and useful for teaching and learning. Even in individual pair comparisons, CodeToon comics were also rated more positively (cf. Fig. 19 in Appendix).

Another important metric is whether CodeToon consistently produces high-quality comics. Along with the general observation that Baseline comics vary in quality while CodeToon provides comics of consistent quality (cf. Fig. 19), responses to two agree/disagree statements from the CSI survey provide support. The two statements are: (1) What I was able to produce was worth the effort I had to exert to produce it; (2) I was satisfied with what I got out of the tool. In both statements (1 as Highly Disagree, 10 as Highly Agree), on average, CodeToon users ((1): M=8.83, SD=1.0; (2): M=8.92, SD=1.3) rated higher than Baseline users ((1): M=8.16, SD=1.5; (2): M=8.42, SD=1.6), suggesting that CodeToon users consistently produced comics of the quality they are satisfied with.

7 DISCUSSION

7.1 Implications & Opportunities

Code-Driven Storytelling. The storytelling in CodeToon differs from prior work that leveraged storytelling in computer programming, as stories in CodeToon are structured around code. Whereas code has traditionally been used to define programmable aspects of stories such as animation and interaction (e.g., Scratch [30], Alice [23]), in code-driven storytelling, code functions as a blueprint for stories: the three logic/control structures (sequential, selection, and iteration logic) in code become the structures of the story and comic. As it preserves the code's abstract structure across story and comic, learners can quickly recognize how they correspond to each other and make sense of code expressions, syntax, and conventions in the natural language and visual language of comics while engaging in the creative storytelling process. As general ideas in this approach will likely work with any programming language, this work opens up exciting opportunities to explore how we can leverage story ideation, auto comic generation, and structure mapping in a similar manner for other programming and computational languages (e.g., math).

Storytelling with Text-based Programming. CodeToon enables a way to learn computer programming through storytelling using text-based programming languages. Learning programming through storytelling has traditionally been done using block-based programming languages and environments [23, 30]. Our computational pipeline for transforming text-based code to story and story to comic opens up a whole new direction to explore. For example, this approach could potentially enable us to teach young students who, due to their age, have first been introduced to coding via block-based programming languages. We could test whether they can learn with text-based programming languages using this approach or whether they can learn comic expressions and then use them to learn corresponding text-based programming expressions. Since some students who learn coding first with block-based programming need to re-learn text-based programming when they grow older, this approach could potentially lessen this gap and provide an additional pathway to learning.

Comics for Computational Languages. The feasibility of our approach means that we can explore how other programming languages and paradigms can also leverage comics. For instance, object-oriented programming is an area where its concepts and code are often presented in terms of real-life equivalents [39]. Comics could benefit students learning object-oriented programming. Other programming paradigms such as functional programming may require identifying abstractions (e.g., pointers) important to them for custom visual vocabulary, which would help expand the set of visual vocabularies and advance our understanding of mental models for programming.

Design Implications. Our work has interesting design implications for various domains. Our user study results suggest that the 1-to-1 mapping can be an effective, useful design when using comics as a complementary representation for code. This confirms the suggestion in the literature on multiple representational systems, which recommends making the mapping between multiple representations clear. For research areas—such as data and stats comics [5, 41]—that also leverage comics in much the same way coding strip does, our insights concerning 1-to-1 mapping and the process of building comic expressions can help them explore a similar direction where they can also explore methods to auto generate comics from languages in their domain (e.g., R programming). Our work also leads to many interesting questions for comic authoring tools and coding tools. For instance, how can comic authoring tools utilize our idea of generating comics from language semantics and computational steps to enhance and diversify the authoring process? How can we design coding tools that offer ways to switch between code and other levels of abstraction (e.g., stories and comics)? How can we leverage such interaction to support various tasks programmers perform, e.g., reading and writing code, and debugging? What representations or abstractions can we support in these coding tools? How can we make their transitions seamless? How should we design these transitions and interactions so that we maximize the benefits of multiple representations?

Visual Programming Environment for Artistic Activities. Using coding strips to teach and learn computer programming is a new and promising approach. Recent work showed that it can enhance student learning and address some challenges in teaching programming [33, 36, 38]. But there is still much work to be done. To ease its adoption, we need a curriculum containing a set of learning activities and guidelines in an accessible form, such as a cheat sheet, so that teachers can quickly reference and apply it to their teaching and lessons. Moreover, understanding the nature and impact of this approach needs to be further investigated. For instance, what separates CodeToon from other visual programming environments like Scratch is that it can host artistic activities such as drawing. Recent efforts to combine art and programming—known as *creative coding*—teach and use programming as the primary medium for creating visual artifacts. While CodeToon also allows users to do this to some extent with comic generation, its drawing canvas opens up opportunities for us to potentially explore a different direction—one that does not center around generating art with programming and allows students to learn computational ideas from artistic activities without having to first learn programming. This leads to several questions: What artistic activities can we develop? How can we incorporate them into teaching and learning programming? How can they improve teaching and learning in CS education?

7.2 Limitations & Future Work

Educational Benefits. While prior work on coding strips demonstrated various learning benefits to using coding strips [33, 36], learning with CodeToon is a different process that needs to be examined for several reasons. First, our findings on pedagogical usefulness are participants' *perceived* usefulness. A rigorous study measuring its impact on learning would provide a more accurate assessment of its usefulness. Second, while our study participants included those with experience in teaching programming and learning as former students, their experience in programming and teaching varied. Finally, while CodeToon is a tool for teachers and students not yet proficient in programming, our user study participants were students with experience in programming. This was intentional, as we placed a higher priority on understanding possible usage scenarios and assessing the accuracy of the mapping between comics and code executions and semantics—which learners with little to no programming knowledge would be less qualified to do. Now that we tested CodeToon, we plan to do a study in the future with students who have little or no prior programming experience.

Quality & Limited Set. For the drawings in the auto generated comics, we used the dataset from the Google's Quick, Draw! game [27], which offers sketches of 345 categories. Since they are quick sketches people drew under pressure in a short time, the quality was mostly sub par. For better quality, we chose the AI correctly-guessed drawings. However, we saw several participants who were not satisfied switch to different objects. We suspect this may have given the impression for some that the tool is not for a professional audience but for a young audience. To address this quality issue and the challenge of having a limited set of categories, we plan to explore how we can leverage machine learning to convert existing pictures into comic-style sketches and then use them in the auto generated comics.

Lack of Options. Another limitation in CodeToon is a lack of support for different design languages and workflows. For instance, because we aimed to express semantics for every line, this led to auto generated comics illustrating code that is not executed. For example, for code inside the `if` block where the conditional expression evaluates to `False`, CodeToon still visualized this code even though it was not executed. To indicate this, some participants suggested changing the opacity of these panels, whereas others suggested not showing them. While some were surprised they were shown (thus 4/5 instead of 5/5 for mapping accuracy on semantics), they recognized that this design would be helpful for teaching. Participants also shared alternative design ideas for auto generated comics—such as including code in the auto generated comic—and ideas for improving the workflow, such as real-time rendering. When a user adds a story to the story template, the comic is immediately rendered (or updated). Bi-directional mapping/rendering idea was also discussed—e.g., if the user edits comics, the story and code update accordingly in real-time. Suggestions like these show that users might desire different designs and workflows depending on the context. Therefore, we plan to implement a system preferences panel where users can customize, e.g., the design of the auto generated comics and rendering behaviors.

Generative Models. One exciting avenue to explore next is generative deep learning models. Recently, large language models such as GPT-3 and Codex have shown impressive performance at language-related tasks, including generating code and translating natural language into code and vice versa. Generative model-based support can be an exciting addition to CodeToon. On user's request, generative conversational AI can generate or fill the gap in stories or code. Recent work showed that it can also generate stories from code or code examples for different programming concepts [34]. Based on the analysis of the drawings on the canvas, it can also provide various creativity support, such as generating new drawings to add to the existing comic and providing design guidance in real-time. For instance, if a user modifies the comic and it loses the relational structure carried over from code, it can notify the user and ask if that is the user's intention. These generative model-based interactions would make the authoring process and learning experience more interactive, creative, and collaborative.

8 CONCLUSION

We introduce CodeToon, a comic authoring tool for facilitating the creation of coding strips by supporting story ideation and enabling auto generation of comics from code. To support this code-story-comic mapping, we used structure mapping theory and developed a visual vocabulary for coding strip. Then we tested whether CodeToon successfully supports the authoring of coding strips and helps generate high-quality comics through a two-part user study. The results of our user and comic evaluation studies show CodeToon not only supports the authoring of coding strips well but also reduces the comic authoring time by a significant amount while producing accurate, informative, and useful coding strips. In addition to contributing CodeToon and the two-part user study, our work lays the groundwork for code-driven storytelling by contributing computational pipeline and design guidelines for effective code-story-comic mapping.

ACKNOWLEDGMENTS

This research was funded by the Learning Innovation and Technology Enhancement (LITE) Grant at the University of Waterloo. The authors also thank our study participants for their participation and reviewers for their feedback and suggestions.

REFERENCES

[1] 2019. Coding Strip. https://codingstrip.github.io/ Accessed: 2021-08-05.

[2] 2019. Pixton. https://www.pixton.com/ Accessed: 2020-01-05.

[3] Shaaron Ainsworth. 2006. DeFT: A conceptual framework for considering learning with multiple representations. *Learning and instruction* 16, 3 (2006), 183–198. https://doi.org/10.1016/j.learninstruc.2006.03.001

[4] Tiago Alves, Adrian McMichael, Ana Simões, Marco Vala, Ana Paiva, and Ruth Aylett. 2007. Comics2D: Describing and creating comics from story-based applications with autonomous characters. *Proceedings of CASA* (2007).

[5] Benjamin Bach, Nathalie Henry Riche, Sheelagh Carpendale, and Hanspeter Pfister. 2017. The emerging genre of data comics. *IEEE computer graphics and applications* 37, 3 (2017), 6–13. https://doi.org/10.1109/MCG.2017.33

[6] Benjamin Bach, Zezhong Wang, Matteo Farinella, Dave Murray-Rust, and Nathalie Henry Riche. 2018. Design patterns for data comics. In *Proceedings of the 2018 chi conference on human factors in computing systems.* 1–12. https://doi.org/10.1145/3173574.3173612

[7] Aaron Bangor, Philip T Kortum, and James T Miller. 2008. An empirical evaluation of the system usability scale. *Intl. Journal of Human–Computer Interaction* 24, 6 (2008), 574–594. https://doi.org/10.1080/10447310802205776

[8] Kirsten Berthold and Alexander Renkl. 2009. Instructional aids to support a conceptual understanding of multiple representations. *Journal of Educational Psychology* 101, 1 (2009), 70. https://doi.org/10.1037/a0013247

[9] Ying Cao, Antoni B Chan, and Rynson WH Lau. 2012. Automatic stylistic manga layout. *ACM Transactions on Graphics (TOG)* 31, 6 (2012), 1–10. https://doi.org/10.1145/2366145.2366160

[10] Erin Cherry and Celine Latulipe. 2014. Quantifying the creativity support of digital tools through the creativity support index. *ACM Transactions on Computer-Human Interaction (TOCHI)* 21, 4 (2014), 1–25. https://doi.org/10.1145/2617588

[11] Sung-Bae Cho, Kyung-Joong Kim, and Keum-Sung Hwang. 2007. Generating cartoon-style summary of daily life with multimedia mobile devices. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems.* Springer, 135–144. https://doi.org/10.1007/978-3-540-73325-6_14

[12] Neil Cohn. 2013. *The Visual Language of Comics: Introduction to the Structure and Cognition of Sequential Images.* A&C Black.

[13] Neil Cohn. 2015. How to analyze visual narratives: A tutorial in Visual Narrative Grammar. (2015). http://visuallanguagelab.com/P/VNG_Tutorial.pdf Accessed: 2021-06-01.

[14] Emily R Fyfe and Mitchell J Nathan. 2018. Making "concreteness fading" more concrete as a theory of instruction for promoting transfer. *Educational Review* (2018), 1–20. https://doi.org/10.1080/00131911.2018.1424116

[15] Mirta Galesic and Michael Bosnjak. 2009. Effects of questionnaire length on participation and indicators of response quality in a web survey. *Public opinion quarterly* 73, 2 (2009), 349–360. https://doi.org/10.1093/poq/nfp031

[16] Dedre Gentner. 1983. Structure-mapping: A theoretical framework for analogy. *Cognitive science* 7, 2 (1983), 155–170. https://doi.org/10.1207/s15516709cog0702_3

[17] Dedre Gentner. 1988. Metaphor as structure mapping: The relational shift. *Child development* (1988), 47–59. https://doi.org/10.2307/1130388

[18] Philip J Guo. 2013. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education.* 579–584. https://doi.org/10.1145/2445196.2445368

[19] Juris Hartmanis. 1994. Turing award lecture: On computational complexity and the nature of computer science. *Commun. ACM* 37, 10 (1994), 37–44. https://doi.org/10.1145/194313.214781

[20] Samuel Ichiyé Hayakawa. 1947. Language in action. (1947).

[21] Richang Hong, Xiao-Tong Yuan, Mengdi Xu, Meng Wang, Shuicheng Yan, and Tat-Seng Chua. 2010. Movie2comics: a feast of multimedia artwork. In *Proceedings of the 18th ACM international conference on Multimedia.* ACM, 611–614. https://doi.org/10.1145/1873951.1874033

[22] DaYe Kang, Tony Ho, Nicolai Marquardt, Bilge Mutlu, and Andrea Bianchi. 2021. Toonnote: Improving communication in computational notebooks using interactive data comics. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems.* 1–14. https://doi.org/10.1145/3411764.3445434

[23] Caitlin Kelleher, Randy Pausch, and Sara Kiesler. 2007. Storytelling alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems.* 1455–1464. https://doi.org/10.1145/1240624.1240844

[24] Nam Wook Kim, Nathalie Henry Riche, Benjamin Bach, Guanpeng Xu, Matthew Brehmer, Ken Hinckley, Michel Pahud, Haijun Xia, Michael J McGuffin, and

Hanspeter Pfister. 2019. Datatoon: Drawing dynamic network comics with pen+ touch interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–12. https://doi.org/10.1145/3290605.3300335

[25] Rhonda G Kost and Joel Correa da Rosa. 2018. Impact of survey length and compensation on validity, reliability, and sample characteristics for Ultrashort-, Short-, and Long-Research Participant Perception Surveys. *Journal of clinical and translational science* 2, 1 (2018), 31–37. https://doi.org/10.1017/cts.2018.18

[26] David Kurlander, Tim Skelly, and David Salesin. 1996. Comic chat. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. ACM, 225–236. https://doi.org/10.1145/237170.237260

[27] Google Creative Lab. 2017. Quickdraw. https://github.com/googlecreativelab/quickdraw-dataset Accessed: 2021-08-14.

[28] Tricia J Ngoon, Joy O Kim, and Scott Klemmer. 2021. Shöwn: Adaptive Conceptual Guidance Aids Example Use in Creative Tasks. In *Designing Interactive Systems Conference 2021*. 1834–1845. https://doi.org/10.1145/3461778.3462072

[29] Kesiev Norimaki. 2011. https://www.kesiev.com/stripthis/. Accessed: 2021-10-17.

[30] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay S Silver, Brian Silverman, et al. 2009. Scratch: Programming for all. *Commun. Acm* 52, 11 (2009), 60–67. https://doi.org/10.1145/1592761.1592779

[31] Sangho Suh. 2019. Using concreteness fading to model & design learning process. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*. 353–354. https://doi.org/10.1145/3291279.3339445

[32] Sangho Suh. 2020. Promoting Meaningful Learning by Supporting Interplay within Abstraction Ladder. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–2. https://doi.org/10.1109/VL/HCC50065.2020.9127251

[33] Sangho Suh. 2022. *Coding Strip: A Tool for Supporting Interplay within Abstraction Ladder for Computational Thinking*. Ph.D. Dissertation. University of Waterloo. http://hdl.handle.net/10012/18318

[34] Sangho Suh and Pengcheng An. 2022. Leveraging Generative Conversational AI to Develop a Creative Learning Environment for Computational Thinking. In *27th International Conference on Intelligent User Interfaces*. 73–76. https://doi.org/10.1145/3490100.3516473

[35] Sangho Suh, Sydney Lamorea, Edith Law, and Leah Zhang-Kennedy. 2022. PrivacyToon: Concept-driven Storytelling with Creativity Support for Privacy Concepts. (2022). https://doi.org/10.1145/3532106.3533557

[36] Sangho Suh, Celine Latulipe, Ken Jen Lee, Bernadette Cheng, and Edith Law. 2021. Using Comics to Introduce and Reinforce Programming Concepts in CS1. In *Proceedings of the 52nd ACM technical symposium on Computer science education*. 585–590. https://doi.org/10.1145/3408877.3432465

[37] Sangho Suh, Martinet Lee, and Edith Law. 2020. How do we design for concreteness fading? survey, general framework, and design dimensions. In *Proceedings of the Interaction Design and Children Conference*. 581–588. https://doi.org/10.1145/3392063.3394413

[38] Sangho Suh, Martinet Lee, Gracie Xia, et al. 2020. Coding strip: A pedagogical tool for teaching and learning programming concepts through comics. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–10. https://doi.org/10.1109/VL/HCC50065.2020.9127262

[39] Tevita Tanielu, Raymond 'Akau'ola, Elliot Varoy, and Nasser Giacaman. 2019. Combining analogies and virtual reality for active and visual object-oriented programming. In *Proceedings of the acm conference on global computing education*. 92–98. https://doi.org/10.1145/3300115.3309513

[40] Bret Victor. 2011. Up and Down the Ladder of Abstraction. http://worrydream.com/LadderOfAbstraction/ Accessed: 2021-11-16.

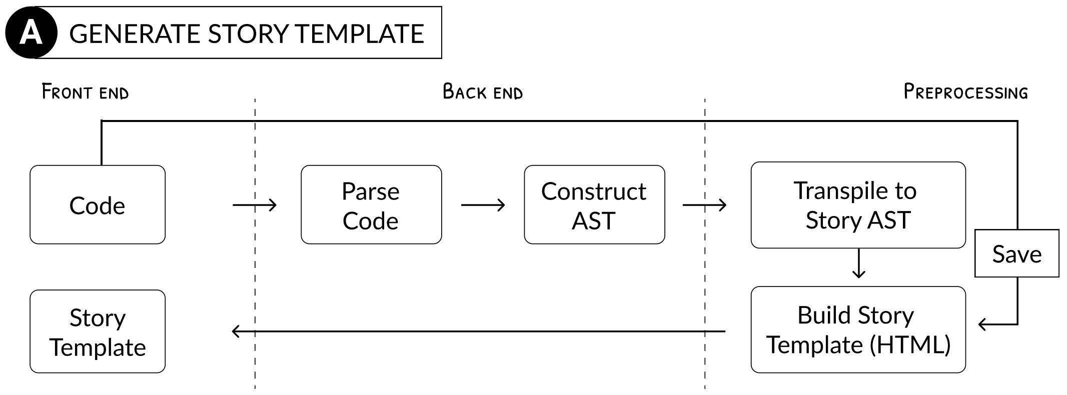
[41] Zezhong Wang, Jacob Ritchie, Jingtao Zhou, Fanny Chevalier, and Benjamin Bach. 2020. Data Comics for Reporting Controlled User Studies in Human-Computer Interaction. *IEEE Transactions on Visualization and Computer Graphics* (2020). https://doi.org/10.1109/tvcg.2020.3030433

[42] Zezhong Wang, Hugo Romat, Fanny Chevalier, Nathalie Henry Riche, Dave Murray-Rust, and Benjamin Bach. 2021. Interactive Data Comics. *IEEE Transactions on Visualization and Computer Graphics* 28, 1 (2021), 944–954. https://doi.org/10.1109/TVCG.2021.3114849

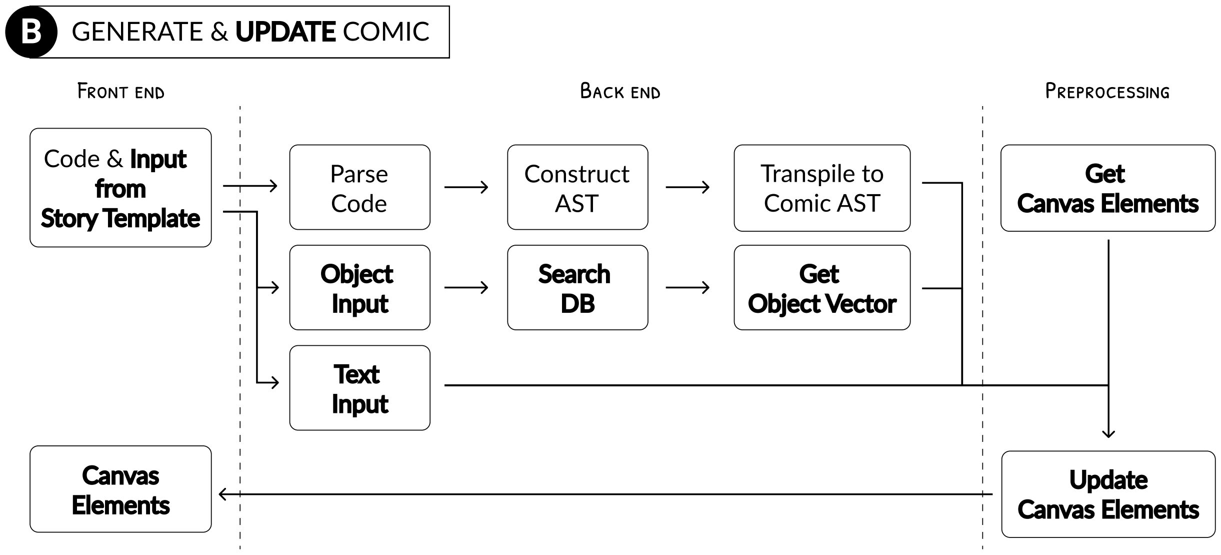
[43] Xin Yang, Zongliang Ma, Letian Yu, Ying Cao, Baocai Yin, Xiaopeng Wei, Qiang Zhang, and Rynson WH Lau. 2021. Automatic Comic Generation with Stylistic Multi-page Layouts and Emotion-driven Text Balloon Generation. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 17, 2 (2021), 1–19. https://doi.org/10.1145/3440053

[44] R Zeeders. 2010. *Comics-comic generation from story content graphs*. Ph.D. Dissertation. Master's thesis, University of Twente, Department of Electrical Engineering, Mathematics and Computer Science.

A APPENDIX

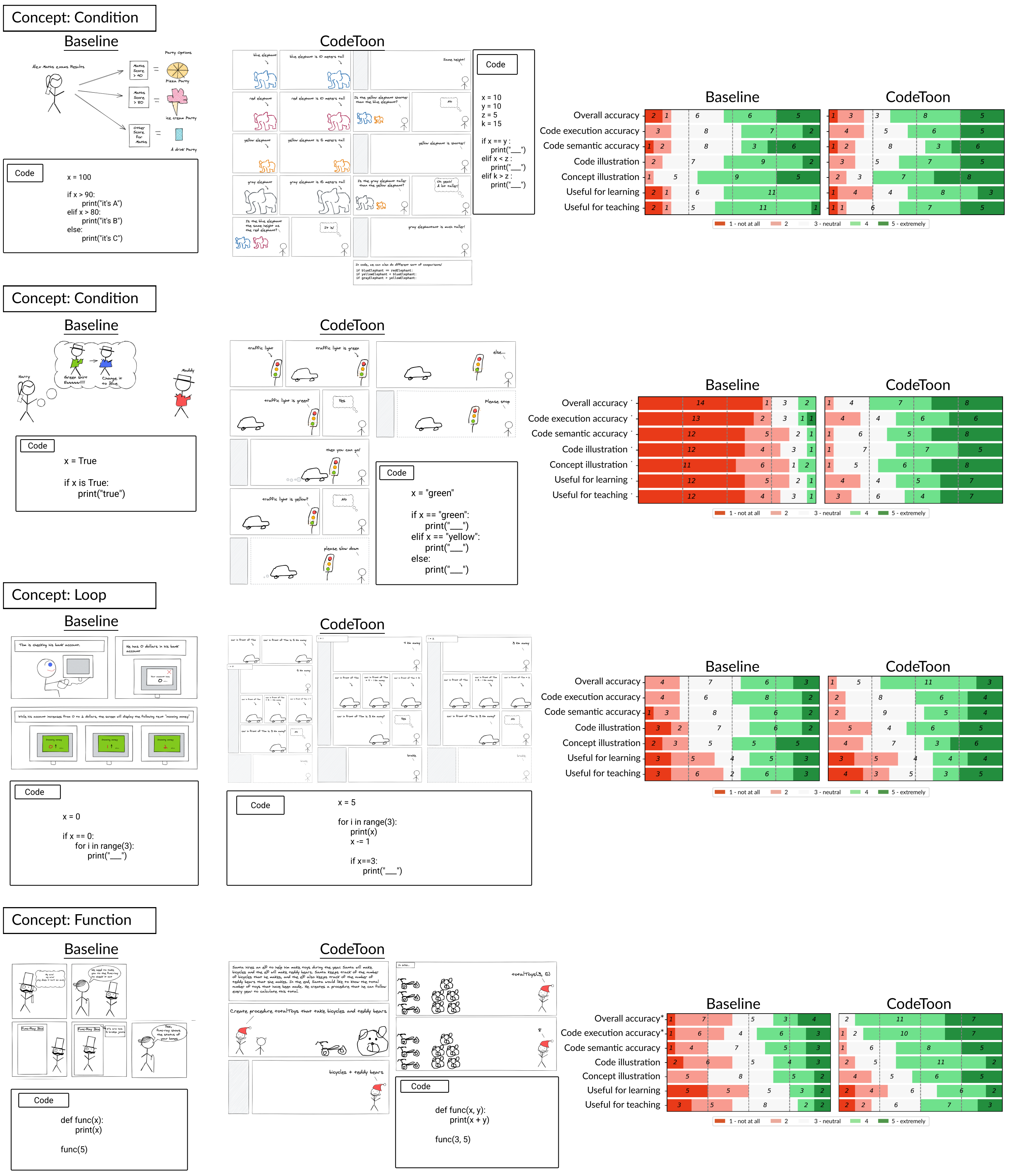


(a) Pipeline for generating story template. When a user clicks the 'generate story' button (Fig. 3C), the code is sent to the back-end and parsed into an abstract syntax tree (AST) in JSON. To build a story template, the program traverses through the code AST (JSON) recursively. As it traverses, it checks node types (e.g., <Assign>) and extracts relevant information to build the story template, which consists mostly of `<input>` HTML tag. Once the story template has been generated, it is sent to the front end to be rendered in the story section (Fig. 3D).



(b) Pipeline for generating and updating comic from story. When a user clicks the 'generate comic' button (Fig. 6), CodeToon collects (1) code that was used to generate the story template and (2) values in the story template and makes an Ajax request. In the back-end, the program parses code and turns it into comic AST, which is essentially an instruction of how comic should be presented graphically (cf. Fig. 2). At the same time, the back-end checks whether any text made reference to the object in the object database. If there is a match, the back-end returns the vector information so they can be added to the comic. Once these information come together, they are sent to the front-end to render the comic in the canvas. As for updating the comic, when a user clicks the 'update comic' button, it follows the same procedure (highlighted in bold), except that code is not sent to the back-end, because we are not generating a new comic. Note that 'Get Canvas Elements' in the pipeline captures the elements currently present in the canvas. This is used to avoid (1) overwriting the entire canvas with new comic elements and (2) placing newly generated comic on top of existing elements.

Figure 18: Implementation of computational pipeline for (a) generating story template and (b) generating and updating comic from story.



Code
```
x = 100

if x > 90:
    print("it's A")
elif x > 80:
    print("it's B")
else:
    print("it's C")
```

Code
```
x = 10
y = 10
z = 5
k = 15

if x == y :
    print("__")
elif x < z :
    print("__")
elif k > z :
    print("__")
```

Code
```
x = True

if x is True:
    print("true")
```

Code
```
x = "green"

if x == "green":
    print("__")
elif x == "yellow":
    print("__")
else:
    print("__")
```

Code
```
x = 0

if x == 0:
    for i in range(3):
        print("__")
```

Code
```
x = 5

for i in range(3):
    print(x)
    x -= 1

    if x==3:
        print("__")
```

Code
```
def func(x):
    print(x)

func(5)
```

Code
```
def func(x, y):
    print(x + y)

func(3, 5)
```

Figure 19: Comparisons of individual pairs according to their concepts. CodeToon comics were rated as being as good or better than Baseline comics in all cases across all measures.