Informatics

FlowTutor: Programmieren mit Flußdiagrammen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Thomas Rößl, B.Sc.
Matrikelnummer 11775192

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Michael Waltl
Associate Prof. Dr.techn. Lado Filipovic

Wien, 11. März 2024

Thomas Rößl   Michael Waltl

Informatics

FlowTutor: Programming Using Flowcharts

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Thomas Rößl, B.Sc.
Registration Number 11775192

to the Faculty of Informatics
at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Michael Waltl
Associate Prof. Dr.techn. Lado Filipovic

Vienna, March 11, 2024

Thomas Rößl Michael Waltl

Erklärung zur Verfassung der Arbeit

Thomas Rößl, B.Sc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 11. März 2024

Thomas Rößl

Danksagung

Ich bin Prof. Michael Waltl und Prof. Lado Filipovic zutiefst dankbar, dass sie mir die Möglichkeit gegeben haben diese Arbeit durchzuführen. Ihre unermüdliche Unterstützung, Anleitung und Geduld während dieses Unterfangens waren wirklich von unschätzbarem Wert.

Ich danke den Teilnehmern an den Evaluierungen für ihr enthusiastisches Engagement, ihr wertvolles Feedback und ihre Bereitschaft, ihre Erfahrungen zu teilen. Ihre Beiträge haben diese Untersuchung erheblich bereichert.

Ohne den Einsatz der vielen Open-Source-Tools und Ressourcen wäre ein um ein Vielfaches höherer Aufwand erforderlich, um eine Anwendung zu entwickeln, wie wir es im Rahmen dieser Arbeit getan haben. Ich möchte allen danken, die zu dieser großartigen Community beitragen.

Meinen Kollegen und Freunden danke ich aufrichtig für ihre unermüdliche Unterstützung und Ermutigungen während dieser Reise.

Acknowledgements

I am deeply grateful to Prof. Michael Waltl and Prof. Lado Filipovic for entrusting me with the opportunity to conduct this thesis. Their unwavering support, invaluable guidance, and patience throughout this endeavor were truly invaluable.

I extend my heartfelt appreciation to the participants of the evaluations for their enthusiastic engagement, valuable feedback, and willingness to share their experiences. Their contributions have significantly enriched this research.

Without the use of the many open-source tools and resources, orders of magnitude more effort would be necessary to develop an application, as we have done in the course of this work. I want to thank everyone who contributes to this amazing community.

To my colleagues and friends, I express sincere gratitude for their unwavering support and encouragement during this journey.

Kurzfassung

In der Programmierlehre setzt der traditionelle Ansatz oft auf syntaxlastige Vorlesungen und Programmierübungen, welche speziell für Personen, welche zum ersten Mal mit einer Programmiersprache konfrontiert werden, besonders herausfordernd sein können. Um den Einstieg in das Programmieren zu erleichtern, werden in dieser Arbeit das Design, die Entwicklung und die Evaluierung einer Anwendung vorgestellt, die darauf abzielt, das Erlernen von Programmierkonzepten durch die Verwendung von interaktiven Flussdiagrammen zu verbessern.

Die Anwendung mit dem Namen "FlowTutor" ermöglicht es Studierenden, Flussdiagramme zu erstellen, welche die Programmierlogik darstellen. Durch das Verbinden von Knoten, die Programmierkonstrukte wie Schleifen, Konditionale und Variablen darstellen, können die Auszubildenden intuitiv und unterstützt durch grafische Darstellung ihre Algorithmen entwerfen und den Ablauf der Ausführung effizienter nachvollziehen und auch verstehen.

Die Definition benutzerdefinierter Knotentypen ermöglicht es, Quellcode für beliebige Programmiersprachen und Konstrukte zu erzeugen.

Programme können in FlowTutor visuell ausgeführt und debuggt werden, wobei die Möglichkeit besteht, schrittweise durch das Programm zu gehen und Variablenzuweisungen zu visualisieren.

Abschließend wird in dieser Arbeit die Benutzerfreundlichkeit und die Arbeitsbelastung der Anwendung mithilfe der System Usability Scale (SUS) und des NASA Task Load Index (TLX) mit einem mixed-methods Ansatz bewertet. Die SUS-Punktzahl und die TLX Ergebnisse in der Evaluierung legen eine gute Nutzerfreundlichkeit für Personen mit wenig oder gar keiner Programmiererfahrung nahe. Zusätzlich haben wir qualitatives Feedback gesammelt, um die weitere Entwicklung von FlowTutor in der Zukunft zu unterstützen.

Abstract

In the realm of programming education, the traditional approach often involves syntax-heavy lectures and code-writing exercises, which can be very challenging, particularly for beginners who have never dealt with programming languages before. To make the first programming steps more efficient for newcomers, in this thesis the design, development, and evaluation of an application aimed at enhancing the learning experience of programming concepts through the use of interactive flowcharts are presented.

The application, named "FlowTutor" allows students to visually construct flowcharts which represent programming logic. By connecting nodes representing programming constructs such as loops, conditionals, and variables, students can intuitively design algorithms and understand the flow of execution.

A templating system allows for defining custom node types, with which source code can be produced for arbitrary programming languages and constructs.

Programs can be executed and debugged visually within FlowTutor with the ability to step through the program and visualize variable assignments.

Finally, the thesis employs a mixed-methods approach to evaluate the usability and workload of the application, using the System Usability Scale (SUS) and the NASA Task Load Index (TLX), respectively. The SUS score and TLX results in the evaluation suggest that the tool is usable for those with little to no previous programming experience. In addition, we gathered qualitative feedback to aid the further development of FlowTutor in the future.

Contents

CHAPTER 1

Introduction

1.1 Motivation

In 2024, the significance of understanding computer programs and possessing programming skills extends far beyond the realm of computer scientists and mathematicians. Across various career paths, these skills have become indispensable. Austria, in particular, has embraced this trend through a multitude of initiatives focused on IT, informatics, and digitalization. Literacy today is synonymous with digital literacy, and Austrian students are required to obtain digital literacy skills and programming fundamentals. This foundational knowledge equips them for the digital landscape they will encounter throughout their lives. *Digital basic education*[1] has become a compulsory subject for Austrian students beginning with the school year 2022/23 [Bunb] and since 2018, Austria has recognized the importance of coding skills by introducing a dedicated apprentice profession dedicated to "Coding". Aspiring coders receive structured training, preparing them for careers in software development [Wir].

Furthermore, a new university focusing on digitalization and digital transformation is planned in Linz [Buna] and a survey from 2019 of joinery companies in 2019 showed, that 85% of respondents regard digitalization as "very important" for their companies in the next five years [Fac19].

Although other paradigms exist, *imperative* programming languages are the most common. Imperative means that the programmer defines the program by writing down statements in a defined sequence, which get executed in this order [GA16]. This gets extended by *procedural* languages where the program is structured around procedures, also known as functions or subroutines. In the yearly report "The State of Developer Ecosystem 2023" [Jet23a] almost all of the most widely used programming languages are at least partially

[1] Digitale Grundbildung

procedural, with the only exceptions being markup- and query languages like SQL and HTML.

To learn a procedural programming language, one has to learn a particular way of problem-solving, which includes the ability to analyze a problem and break it down into discrete steps, which can be executed by a computer. One of the core challenges, when teaching students who are new to programming, is often that the technical overhead of learning to use text editors and compilers [CM05], as well as learning the syntax of a particular language [GM15], make the learning curve to get started rather steep. Thus, the use of graphical languages [GM15] and visualization techniques [BK00] is considered a massive help for beginners to focus on the abstract problem-solving aspect of programming.

1.2 Goals

To support the ideas outlined above, a flowchart-based software tool has been developed within the scope of this work. There are plans to use this tool during introductory programming courses at the *Institute for Microelectronics* at TU Wien. During the development special focus was placed on maximizing learning efficiency, particularly to newcomers in software development and programming. Finally, this tool has been evaluated by students in the bachelor program of electrical engineering, examining usability metrics and qualitative aspects regarding the motivation of the students to use such a tool and to gather feedback for further development.

1.3 Historical Background

1.3.1 Programming Education

Computer science did not emerge as an independent field of study from the outset. Its origins can be traced back to the early days of computing when the focus was primarily on developing and utilizing electronic computers for specific applications.

In 1937, Cambridge University established the "Mathematical Laboratory" [Cro92] as an early initiative to explore electronic computing technologies. In 1953, the same institution marked a significant milestone with the creation of the first academic degree program in the field, known as the "Diploma in Numerical Analysis and Automatic Computing" [Spa01].

At the TU Wien, the first course on computing started in 1953 and the first informatics curriculum was introduced in 1970 [TU b]. Today, Informatics is the second most popular field of study at TU Wien, with a 19.4% share of all registered students [TU a]. In all of Germany, it is even the most popular in the STEM (Science, Technology, Engineering and Mathematics) field, with a 23.9% share of all students [Sta].

1.3.2 Graphical Programming

For this work, we define graphical programming systems as any programming environment, where the instructions are defined through graphical representations, rather than pure textual source code alone. Other terms used in literature are *Visual Programming* [RM17], and *Iconic Programming* [Cal92].

The idea of graphical programming environments is at least as old as graphical user interfaces (GUIs) for personal computers are available [GT84]. In his 1975 Ph.D. dissertation, Smith stated that the goal is "[...] to develop a computer system whose representational and processing facilities correspond to and assist mental processes that occur during creative thought." [Smi75].

Graphical programming languages have encountered significant challenges, when used as a general-purpose language, because visual metaphors quickly become overwhelming if all implementation details must be visually present [BM95]. The use of diagrams and flowcharts plays a significant role in software development processes, but rather as a means for abstract conceptualization and documentation purposes, rather than for use in concrete implementations. Standardized as UML (Unified Modelling Language) [BM99] graphical diagrams are used throughout the software engineering process.

In the context of programming education, the requirements are different. Students need only to solve very constrained problems, which can be more easily represented graphically, and there have been numerous evaluations which show the high efficiency of graphical programming for educational purposes [GM15] [Gaj18] [CWHH04].

1.3.3 Flowcharts

A flowchart is a graphical representation of a workflow or process that illustrates the steps, logic, and control of a program or system. It visually depicts how different actions or tasks are connected and the order in which they occur.

One of the earliest formalizations of this concept was introduced at the *Annual Meeting of the American Society of Mechanical Engineers* by Frank and Lillian Gilbreth in 1921 [GG21].

The method involves representing a series of operations or steps in a process using symbols, connected by arrows. Different symbols denote different activities, such as operations, inspections, transportation, and delays. The Gilbreths aimed to improve efficiency and eliminate unnecessary steps in work processes by providing a visual tool for analyzing and optimizing workflows. Their work laid the foundation for modern process mapping and flowcharting techniques widely used in industries today.

Based on their work, the American Society of Mechanical Engineers (ASME) released a standardization "Operation and Flow Process Charts" in 1947 [Spe47]. An example of a typical operation process chart is shown in Figure 1.1.

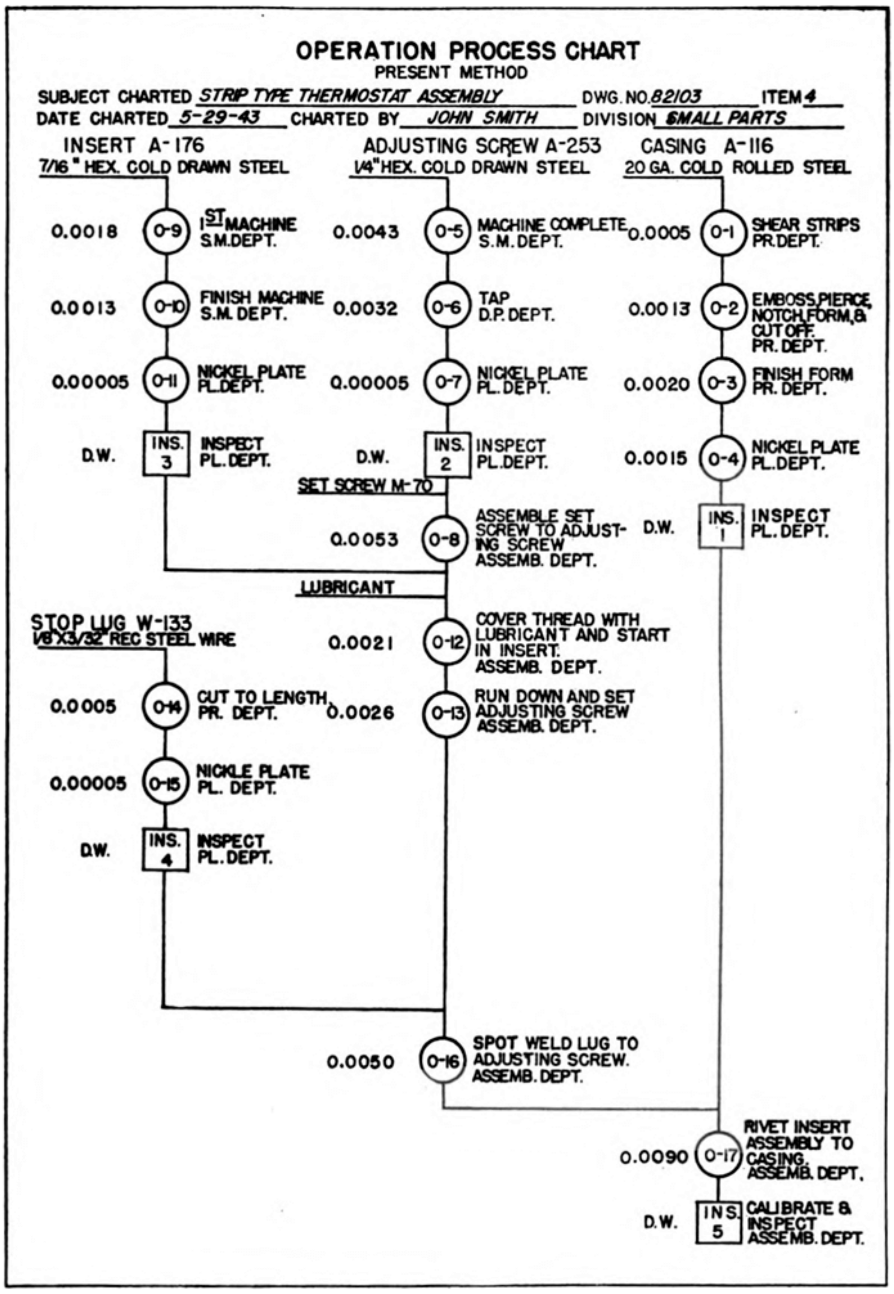


Figure 1.1: ASME operation process chart from 1921, originally published in [GG21].

Flowcharting in the realm of computing goes back to Goldstine and von Neumann at Princeton [Cha70], who published the systematic use of flowcharts for the first time in 1947 [GvN47]. As an example of an algorithm represented as a flowchart, see Figure 1.2.

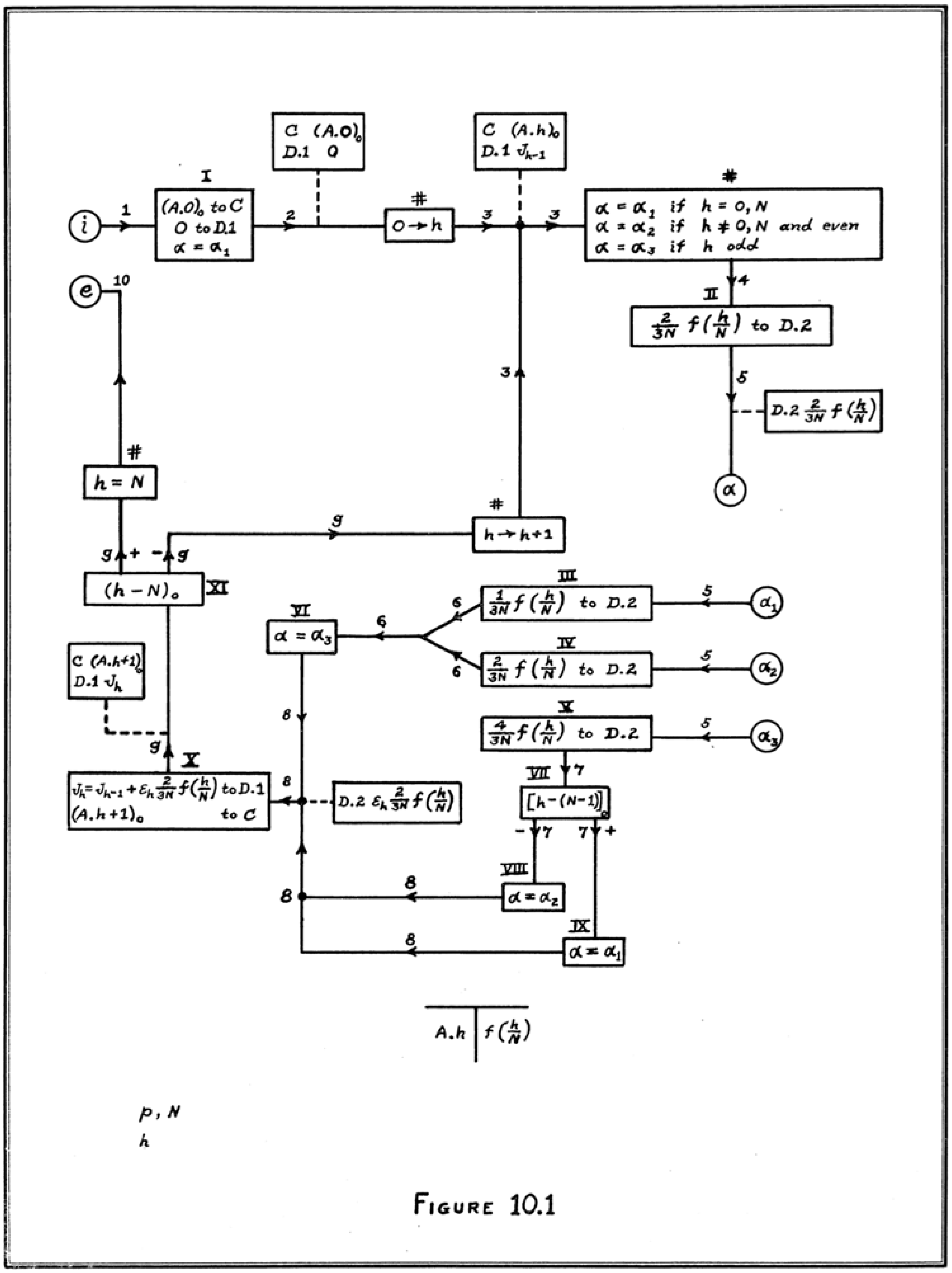


Figure 1.2: Flowchart by Goldstine and von Neumann describing the evaluation of an integral, 1947, originally published in [GvN47].

After the publication of Goldstine and von Neumann's work, companies and organizations adopted the use of flowcharts, often with adaptations and additions of their own [Cha70].

The American Standards Association (ASA), now known as American National Standards Institute (ANSI), started the standardization of flowcharting techniques in the early 1960s, published the first standard in 1963, and kept revising it up to 1970 [ans70].

During that same time, the International Organization for Standardization (ISO) started work on its own standard. The Technical Committee ISO/TC 97, *Computers and information processing* adopted a draft recommendation, which was accepted as an official recommendation in 1969 [iso69]. This recommendation was transformed into an official ISO standard in 1975 [iso73], which in turn was superseded by ISO 5807 "Information processing - Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts" in 1985 [iso85]. An example of a program flowchart according to the ISO 5807 standard is provided in Figure 1.3. This is the standard still in use today.

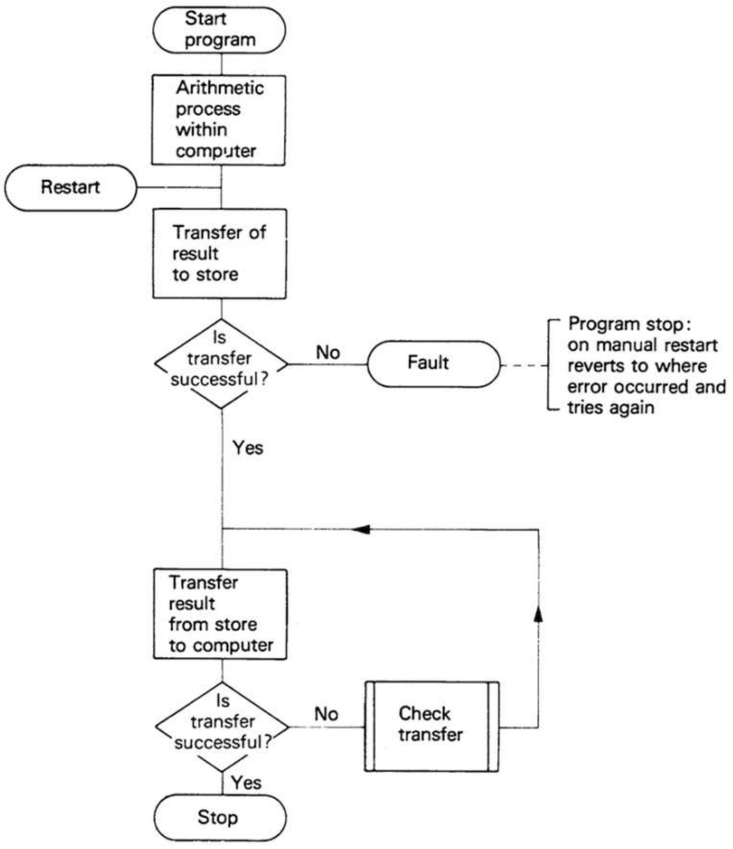

Start program
Arithmetic process within computer
Restart
Transfer of result to store
Is transfer successful?
No
Fault
Program stop: on manual restart reverts to where error occurred and tries again
Yes
Transfer result from store to computer
Is transfer successful?
No
Check transfer
Yes
Stop

Figure 1.3: A program flowchart based on standard ISO 5807, originally published in [iso85].

CHAPTER 2

Related Work

As discussed in the introduction, we mainly concentrate on work, which focuses on programming education as their main goal. Most other graphical programming systems are not general-purpose solutions and only try to solve specific constrained problems.

Over the years, there have been a number of graphical programming systems, mainly aimed at education. Some have used the same visual framework of flowcharts as we have with FlowTutor, but there are different approaches as well. As examples for other visualization systems, we cite some systems using blocks as their main metaphor.

2.1 Flowchart Based

2.1.1 BACCII/BACCII++

BACCII [Cal92] and its successor *BACCII++* [CBH97] were developed in 1992 and 1997, respectively, by Calloni at Texas Tech University. See Figure 2.1 for an exemplary program. BACCII uses custom icons as its flowchart nodes, rather than the ISO 5807 symbols.

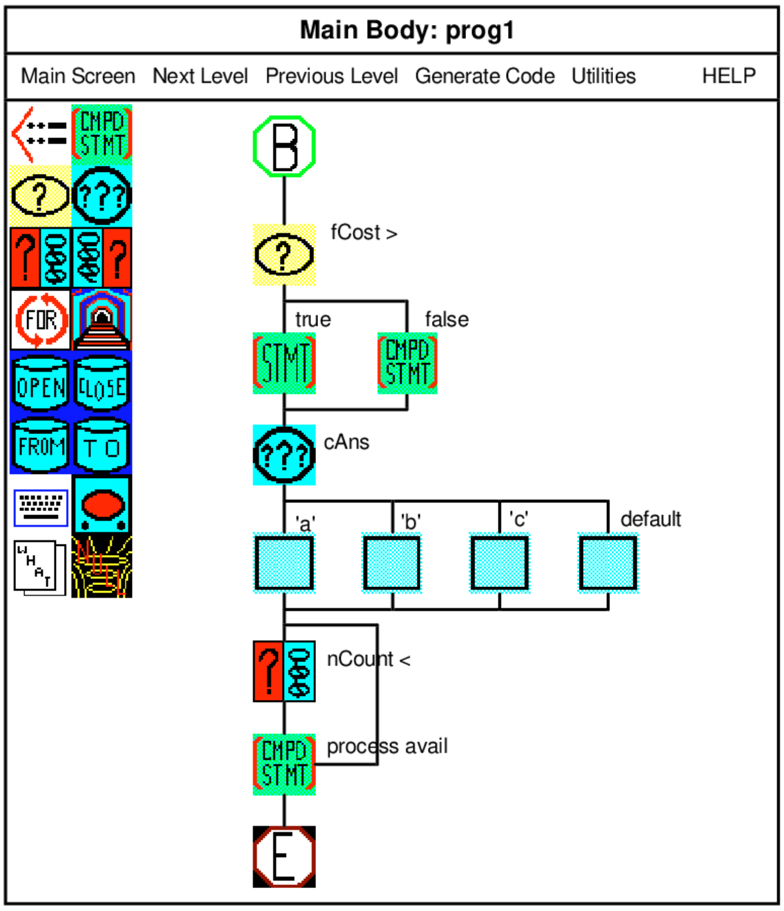


Figure 2.1: BACCII++, originally published in [Cal92].

In their evaluation, they found that "[...] the BACCII++ students showed higher comprehension of the C++ syntax" [CBH97], when compared to a group which was not using the program.

Because it was developed between 1992 and 1996 for the versions of Microsoft Windows available at the time and there has not been any further development, it is not possible to easily execute BACII or BACCII++ on modern machines.

2.1.2 RAPTOR

RAPTOR (Rapid Algorithmic Prototyping Tool for Ordered Reasoning) [CWHH04] was developed in 2004 by Carlisle at the United States Air Force Academy. See Figure 2.2, for an example of its GUI. It was "designed specifically to address the shortcomings of syntactic difficulties and non-visual environments" [CWHH04]. The flowchart nodes are based on the ISO 5807 standard.

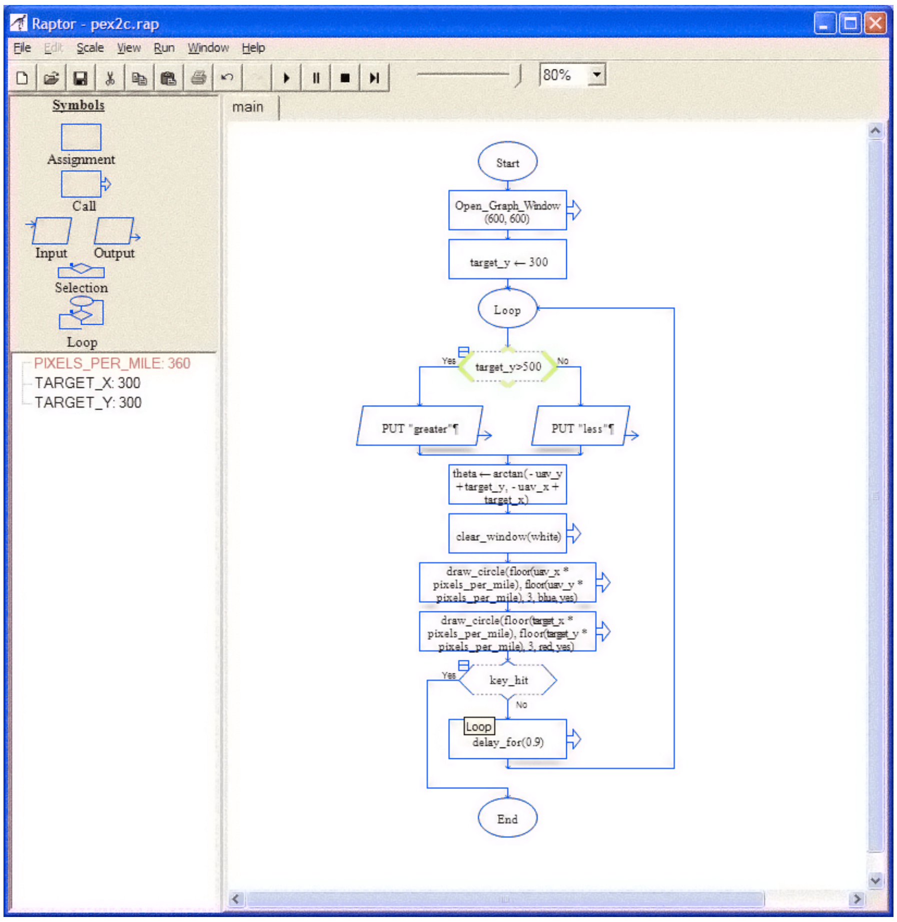

Raptor - pex2c.rap
File
Edit
Scale
View
Run
Window
Help
80%
Symbols
Assignment
Call
Input
Output
Selection
Loop
PIXELS_PER_MILE: 360
TARGET_X: 300
TARGET_Y: 300
main
Start
Open_Graph_Window (600, 600)
target_y ← 300
Loop
Yes
target_y>500
No
PUT "greater"¶
PUT "less"¶
theta ← arctan(- uav_y + target_y, - uav_x + target_x)
clear_window(white)
draw_circle(floor(uav_x * pixels_per_mile), floor(uav_y * pixels_per_mile), 3, blue, yes)
draw_circle(floor(target_x * pixels_per_mile), floor(target_y * pixels_per_mile), 3, red, yes)
Yes
key_hit
No
Loop
delay_for(0.9)
End

Figure 2.2: "RAPTOR in action". Originally published in [CWHH04].

In their evaluation, they found that students preferred the visual approach when given the choice and "[...] that students using RAPTOR who entered the course with a much lower incoming GPA outperformed students with a higher incoming GPA using Ada or MATLAB." [CWHH04].

RAPTOR is open-source and written in a combination of Ada, C#, and C++. Originally developed for Windows, there is a version utilizing the cross-platform Avalonia framework.

2.1.3 Flowgorithm

Flowgorithm [Coo15] was developed in 2015 by Cook at California State University. For an example of its user interface see Figure 2.3. Flowgorithm also bases its flowchart nodes on the ISO 5807 standard. Flowgorithm can convert the flowcharts into a comprehensive list of common programming languages and has a built-in debugger, similar to FlowTutor.

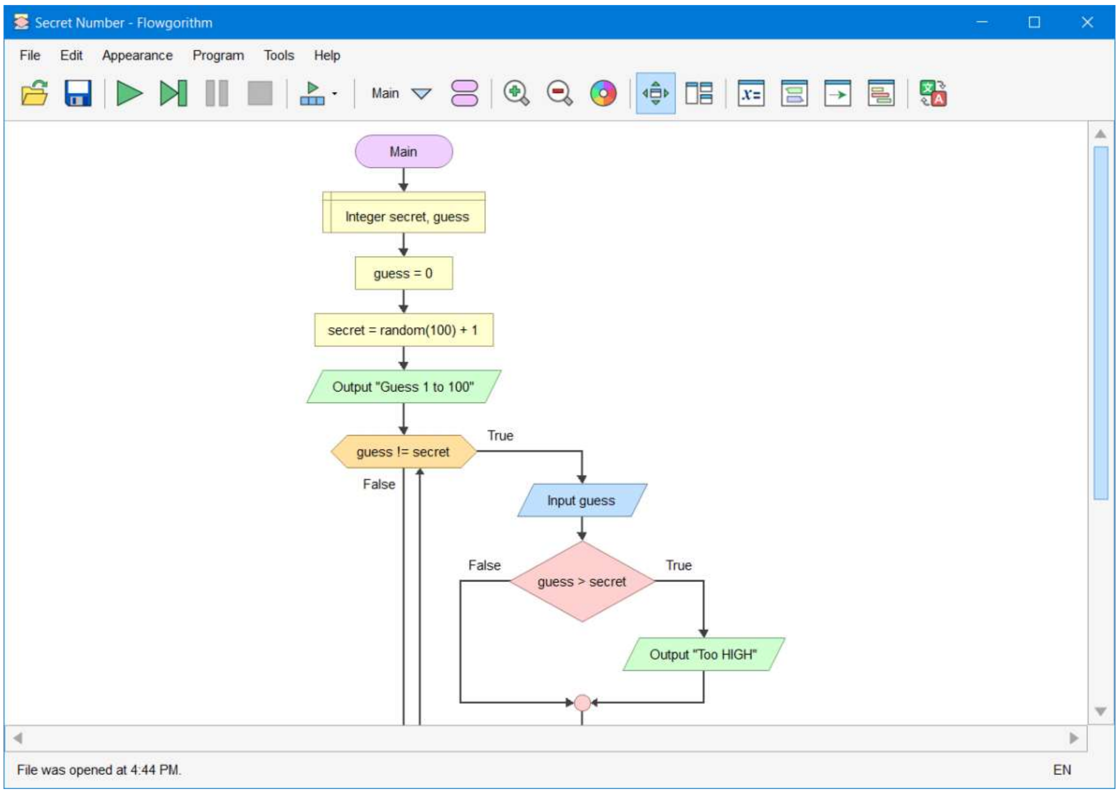


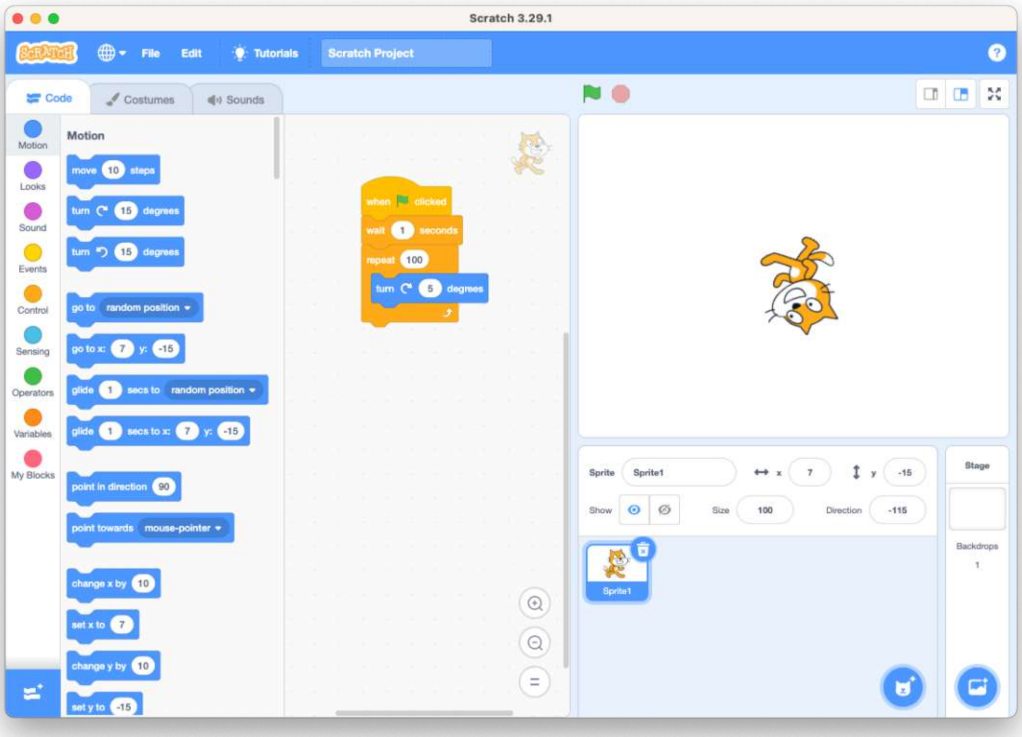
Figure 2.3: Flowgorithm, originally published in [Coo15].

While there was no formal evaluation in the original work, Cook noted that "Student reaction to the software, as well as comments from the website, have been extremely positive" [Coo15]. In a later paper, Gajewski concluded, that "Flowgorithm enabled to distinguish between programming (creating an algorithm) and coding (representing an algorithm in a particular programming language) and concentrate on algorithms and programming" [Gaj18].

As a proprietary piece of software there are some concerns with long-term reliability and possibilities to influence future development, should there be changing requirements in the future. Flowgorithm is written using Microsoft's ".NET" framework and as such can only be run on Windows, which hinders its use in a diverse classroom, where students might use MacOS or Linux.

2.2 Block Based

2.2.1 Scratch



```
when 🏁 clicked
wait 1 seconds
repeat 100
    turn ↻ 5 degrees
```

Figure 2.4: Scratch GUI.

Scratch [MBK+04] was developed in 2004 by Maloney et al. at MIT (Figure 2.4). It is aimed at children aged 8-14 and provides a user-friendly platform for creating interactive stories, games, and animations, making it particularly popular among beginners, children, and early educators.

Users build programs by dragging and dropping visual blocks which represent code structures. These blocks snap together like puzzle pieces. Through these commands, characters or objects are controlled and respond to various events and user inputs.

In an evaluation during Harvard Summer School's "Computer Science S-1: Great Ideas in Computer Science" program, Malan et al. found that Scratch was perceived positively by students, with 75% feeling it was a positive influence and only 8% feeling it was a negative influence [ML07].

2.2.2 Snap!

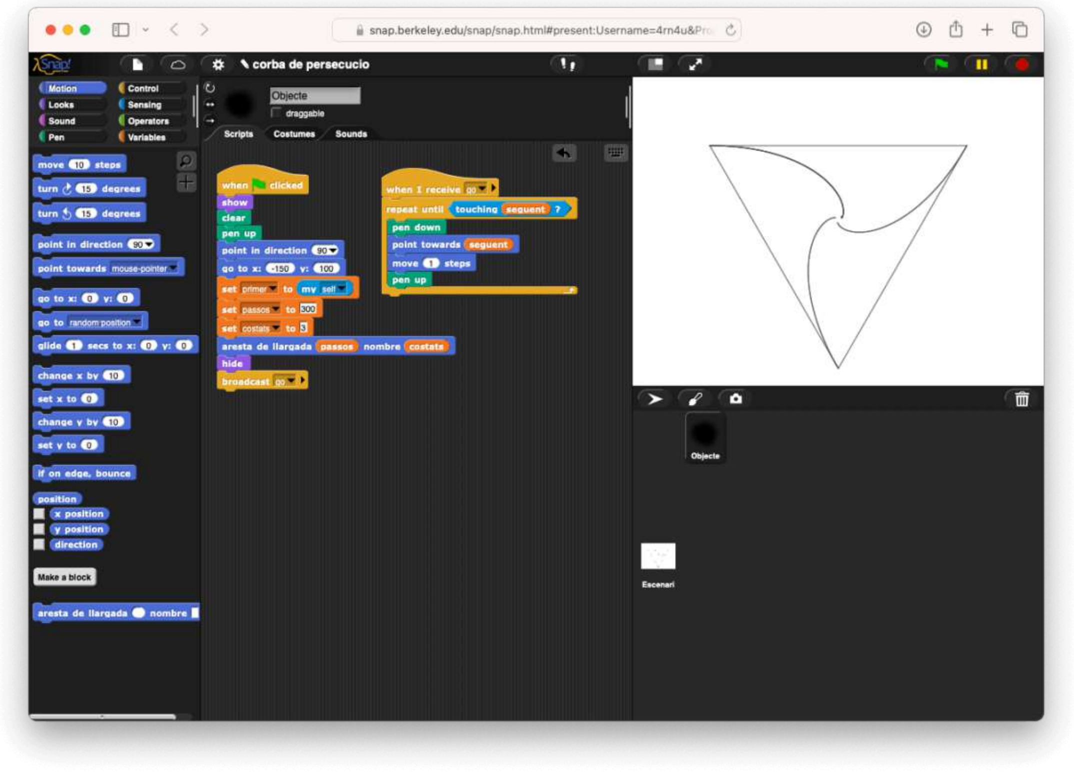


Figure 2.5: "Snap!" web-based GUI.

"Snap!" formerly known as BYOB (Build Your Own Blocks) [HM10] was developed in 2010 by Harvey and Mönig at UC Berkeley. See Figure 2.5 for an example of its web-based user interface. "Snap!" is a reimplementation of Scratch with an enhanced feature set for high school and university students.

"Snap!" extends its support to learners aged 14-20 by introducing named procedures, thereby enabling recursion, treating procedures as data to facilitate higher-order functions, incorporating structured lists, and elevating sprites to first-class objects with inheritance capabilities [HGB^{+}13].

CHAPTER 3

Design and Development

3.1 Requirements

For the gathering of requirements, we used the *Persona* technique [SHM$^+$12]. Terms like “user” are vague and can mean different things to different people [PA10]. By defining fully formed fictitious characters, we can focus the process on requirements for real people and all participants have a better understanding of the goals which should be achieved [SHM$^+$12].

The requirements themselves were formulated as user stories. Following an agile development paradigm, the user stories do not specify every detail of the application but rather focus on the value they bring to the users [Coh04]. These stories are not static entities which get defined in a step in the process and then are fixed in place; rather but they are allowed to evolve with every iteration of the application. There may be details which get added, or large stories which get split into many smaller ones [Coh04].

For this work, we defined three personas and created a list of user stories based on these personas. In the following sections, we list the persons first, followed by deriving the user stories.

3.1.1 Personas

Programming Instructor

Name: Associate Prof. Dipl.-Ing. Dr.techn. Stefanie Schneider

Age: 42

Background:

Prof. Stefanie Schneider is an Associate Professor of Electrical Engineering with a passion for nurturing the next generation of engineers. Her journey into academia began with a deep fascination for the intricate world of circuits and systems. Stefanie earned her Ph.D. in Electrical Engineering, specializing in signal processing and control systems. Throughout her academic career, she discovered the transformative power of demystifying complex concepts, especially when it came to teaching the fundamentals of programming to aspiring electrical engineers.

Stefanie's research endeavors have often intersected with the practical application of programming in the context of hardware design, embedded systems, and control algorithms. Drawing from this rich experience, she transitioned seamlessly into a teaching role, driven by the desire to bridge the gap between theoretical knowledge and hands-on application for her students.

She holds an introductory C-programming lecture with exercises for electrical engineering students.

Beginner Programmer

Name: Mario Egger

Age: 20

Background:

Unlike many of Mario's peers, he did not grow up with a computer at his fingertips but rather with a soldering iron in hand.

During school, where his strengths lay in physics and mathematics, Mario found a passion for understanding the fundamentals of electrical systems. The allure of tinkering with wires and discovering how components interacted became the highlight of his afternoons. While his friends were immersed in the digital world, Mario was more at home with analog devices and hands-on experiments.

Choosing Electrical Engineering for his university studies, Mario faced the challenge of integrating programming into his skill set. Unlike some of his classmates who had been coding since their early teens, he approaches programming languages with a fresh perspective. The world of algorithms and code structures is a new territory, and Mario embraces the challenge with a determination to understand the digital aspects of his field.

To help him with his understanding he decided to attend the lectures of Prof. Schneider.

Intermediate Programmer

Name: Lena Ziegler

Age: 19

Background:

Lena discovered her love for technology through a childhood fascination with gadgets. Growing up in a household where discussions about the latest tech trends were as common as family dinners, Lena's interest in technology was sparked during her early teenage years.

At the age of 14 Lena attended a coding camp, where she was introduced to the world of Python. Over high school summers, Lena delved into online coding tutorials, playing with languages like JavaScript and exploring web development. While her friends were absorbed in social media, Lena was building small websites and experimenting with interactive features.

Despite her knowledge of the basics, Lena still struggles with understanding complex algorithms and data structures. She also attends the lecture of Prof. Schneider.

3.1.2 User Stories

Based on these personas we gathered the following initial user stories and grouped them into categories:

Prerequisites

- Stefanie uses Python frequently in her research and students also learn Python. For ease of maintaining the application, FlowTutor should be written in Python.
- To maintain the ability to improve the application in the future and assure availability, FlowTutor should be licensed with an open-source license.
- Many students use operating systems other than Windows. FlowTutor should be able to run cross-platform without configuration necessary for the end-user.

Program Creation

- Mario has to solve an assignment involving variables. He needs to draw a flowchart which includes a variable declaration node.
- Based on the previous story, Mario needs to draw a flowchart which includes a variable assignment node.

- Lena needs to solve an assignment involving even and odd numbers. For that, she needs to draw a flowchart which includes a decision node (“if-else” in C-syntax).
- Lena needs to solve an assignment involving an arbitrary length series of numbers. For that, she needs to draw a flowchart which includes a loop node (“while” in C-syntax).
- Based on the previous story, Lena wants to simplify the construct with a “for-loop”, this also helps to more easily convey the meaning of the program.
- Stefanie makes an example program and wants to clarify a step in the algorithm. To this end, she wants to be able to add comments to the flowchart.
- Lena encounters a scenario, where she needs to make several nested conditional statements to cover different cases. She wants to simplify this construct with a node which represents a “switch-case” instruction.
- Stefanie wants to demonstrate an algorithm, which requires a statement, not covered by the available node types. She needs a node, where she can insert arbitrary code.
- Mario needs to solve an assignment, where he must define a function. He needs to be able to define a function through a separate flowchart.
- Based on the previous story, Mario needs to be able to call his defined function in the main function.
- Stefanie wants to clarify the syntax of C by comparing it with the flowchart-representation. She needs a way to see the resulting source code.

Program Execution and Debugging

- Mario has problems understanding an algorithm. Going through a completed example, step-by-step in the flowchart would aid his learning process.
- Based on the previous story, Mario needs to be able to see the variable assignments, so he can understand the intermediate steps.
- Mario is debugging a program including pointers, he needs to be able to see the values the pointers refer to, not only their memory addresses.
- With the variable assignments alone, it is difficult for Mario to understand complex program states. He would benefit from a visual representation of pointer assignments, arrays, etc.
- Variable scope is a difficult concept for Mario to grasp, the visualization should include a visual representation of it, to help Mario.

- When stepping through a program, Lena always needs to step through a long loop with many steps, before she gets to the position in the program, in which she is interested. The ability to set breakpoints would make her life easier.
- Mario has created a program which prints some textual output to the console. The application needs a way to show this output to Mario.
- Lena's latest program has a bug which leads to a crash. For easier debugging, she needs to see the error messages generated by the compiler and during program execution.
- Lena has created a program, which displays all prime numbers below 100. She wants the ability to vary this limit, without having to change the program source every time. The application should facilitate user input.

Project Management

- Mario tries to solve an assignment but hits a roadblock. He decides to throw away his current attempt and start from a clean slate. FlowTutor needs to support starting fresh.
- Stefanie has created a half-finished example, which she wants her students to complete as an exercise. To this end she needs to save the project to a file, to be able to distribute it.
- Based on the previous story, Lena needs to be able to load this file, to keep working on it.
- Lena realizes that she can create a program for a calculation, which would be useful to her outside of the lecture. She wants to export an executable version of the program for her own use.

Settings

- Stefanie works a lot on FlowTutor projects and sometimes the arrangement and sizing of the User Interface (UI) elements get in her way. She wants a way the customize it by rearranging and resizing the elements.
- Lena likes to work late into the night, the white background of FlowTutor disrupts her preferred dark surroundings. She wants the ability to use a dark color scheme for the UI.

3.2 Tooling and Libraries

There are plans to use FlowTutor as a teaching aid in introductory programming courses at the *Institute of Microelectronics* at the TU Wien. This is reflected in the choice of tools and libraries used, as we want to facilitate an easy path to take over the project in the future. Further development should not be hindered by licensing issues or steep learning curves.

3.2.1 Version Control

Git was chosen as the version control system for the project, as it has become the de facto standard for software development in recent years, with 87% of developers responding of regularly using it in "The State of Developer Ecosystem 2023" by JetBrains [Jet23b]. In the initial development stages, the code was hosted in a repository at the *Institute of Microelectronics* Gitea-Instance[1] but was later relocated to a publicly available GitHub repository[2] with a LGPL-3.0 license, to make it open-source and accessible to a wider audience.

To maintain a consistent style in the commit messages, we followed the *Conventional Commits* [MPSD19] specification for commit messages.

New features were developed in feature branches and merged into the `development` branch. The `master` branch contains code intended for release. For more information on the deployment process, see Section 3.5.

3.2.2 Programming Language

The choice of programming language was Python. While it is not the most widely used language for application development [Jet23a], Python is the second most used programming language in general and at the *Institute of Microelectronics*. It is also taught in the introductory programming courses, where FlowTutor should be utilized. Another advantage of Python is the availability of the *Python Package Index*[3] as a means of distributing the application, which removes the need to maintain packages for different OS platforms. Python is also consistently ranked as one of the languages which is easiest to learn, so students would be able to pick it up quickly to further contribute to development. C++ has a much steeper learning curve before one can be proficient in writing useful and usable code.

[1] `https://tea.iue.tuwien.ac.at`
[2] `https://github.com/thomasroessl/FlowTutor`
[3] `https://pypi.org`

3.2.3 Code Style

To ensure a consistent coding style, we opted to integrate *flake8*[4] into our Continuous Delivery (CD) workflow. It analyzes the source code for style violations, using a ruleset based on *PEP 8 "Style Guide for Python Code"*[vRWC01].

While Python is not a statically typed language, since Python 3.5 it supports the use of *type hints* [Pyta]. Type hints in Python are annotations used to specify the expected data types of variables, function parameters, and return values. Type hints are not enforced by the Python interpreter but serve as a form of documentation and can be used by static type checkers. We choose to enforce a consistent use of type hints throughout the application, by integrating *mypy*[5] into the CD workflow. Through static typing, an IDE can provide better features like code completion and we can detect many programming errors early, which saves time debugging and reduces bugs.

For more information on the CD workflow, see Section 3.5.

3.2.4 GUI Framework

There is no platform-independent standard GUI framework which is part of the standard Python installation in its entirety. We considered several different options:

- Using system native GUI facilities or writing our own:
  This was dismissed early on, as the requirement for platform independence would force us to develop and maintain several different packages for utilizing the facilities on the various OS platforms, or to write our own GUI toolkit, which would go far beyond the scope of this work.
- tkinter[6], the standard Python interface to the Tcl/Tk GUI toolkit:
  We considered tkinter, because of its simplicity and because it is included with the standard Python installation. While the library itself is part of the installation, the Tcl/Tk toolkit is not available on every system. This would introduce a complication, where users have to install Tcl/Tk before they can use FlowTutor. We want to avoid such prerequisites wherever possible.
- PyQt[7], a library with Python bindings for the Qt application framework:
  Qt is a very large and complex framework, which would give us many possibilities in the development of FlowTutor. Its complexity also gives it a steep learning curve and makes it a heavy-weight in terms of file size for distribution. As we wanted the project to be easily taken over by new developers, we decided to explore other options.

[4]`https://github.com/pycqa/flake8`
[5]`https://www.mypy-lang.org`
[6]`https://docs.python.org/3/library/tkinter.html`
[7]`https://www.qt.io/product/framework`

- Kivy[8], an open-source app development framework:
  Kivy supports traditional desktop-style applications, but its focus is on cross-platform touchscreen solutions, which is reflected in its default GUI elements. As FlowTutor is intended as a desktop application, at least in its current form, we found it more appropriate to use another framework instead.

- DearPyGui[9], a cross-platform graphical user interface toolkit:
  DearPyGui is based on Dear ImGui[10], a "Bloat-free Graphical User interface for C++ with minimal dependencies". Despite its simplicity, it also supports drawing custom graphical elements on a canvas-like drawing area. It is licensed with an *MIT License*, which permits its use with almost no restrictions.

We decided to use DearPyGui due to its simplicity and light-weight nature. If new developers take over the project in the future, they should have an easy time understanding the project as a result.

3.2.5 Utility Libraries

Since we aimed to keep our development effort on features that directly benefit the users of *FlowTutor* and to avoid "reinventing the wheel", we used several utility libraries for sufficiently complex tasks. While doing so, we always weighed their usefulness, with the burden of using external code. We also took care, only to use actively maintained projects, with the criterion being, that there should have been a release during the last 24 months.

In the following list, we explain the reasoning behind the use of the external libraries:

- appdirs[11], "a small Python module for determining appropriate platform-specific dirs, e.g. a 'user data dir'.":
  Every OS platform has different default paths for user data. When it comes to supporting many different Linux distributions, which can differ amongst themselves, it becomes infeasible, to try to maintain consistent functionality.

- blinker[12], "provides fast & simple object-to-object and broadcast signaling for Python objects.":
  The development of a signaling library was not in the scope of this project. We used a robust and well-tested solution instead.

[8]`https://kivy.org`
[9]`https://github.com/hoffstadt/DearPyGui`
[10]`https://github.com/ocornut/imgui`
[11]`https://github.com/ActiveState/appdirs`
[12]`https://blinker.readthedocs.io/en/stable/`

- dependency-injector[13], a Dependency Injection framework:
  For similar reasons as the previous library, we also chose not to develop a custom Dependency Injection framework.
- Jinja2[14] is“a fast, expressive, extensible templating engine. Special placeholders in the template allow writing code similar to Python syntax.”:
  The development of a custom templating language was not feasible within the scope of this project.
- pygdbmi[15], a library to interact with the machine interface of the GNU Debugger (GDB):
  We tried using custom code for all communication with the GDB process but ran into many edge cases with race conditions and character encoding faults, so we opted to use this library instead.
- Shapely[16], a library for “manipulation and analysis of geometric objects in the Cartesian plane.”:
  There are many edge cases and numeric complexities when developing functions for geometric calculations. To avoid “reinventing the wheel”, we chose to use a well-tested and established solution.

3.2.6 Testing

The tests are written using the *pytest*[17] framework. Coverage reports are generated with *pytest-cov*[18]. The *tox*[19] tool is used locally to run the test cases with different Python versions. At the time of writing the tested versions are 3.9, 3.10, 3.11 and 3.12.

Our main focus, when writing the test cases was the functionality of the `Flowchart` classes, to ensure consistent results when adding and removing nodes of different types, such as loops and conditional statements, which might split the execution path into multiple branches.

The other major collection of test cases checks the results of the code generator, to ensure that the generated code matches the expected result.

For more information on Testing, see Section 3.4.

[13]https://github.com/ets-labs/python-dependency-injector
[14]https://jinja.palletsprojects.com
[15]https://github.com/cs01/pygdbmi
[16]https://github.com/shapely/shapely
[17]https://docs.pytest.org
[18]https://github.com/pytest-dev/pytest-cov
[19]https://tox.wiki/en/4.11.4/

3.2.7 Continuous Integration

For the choice of Continuous Integration (CI) and Continuous Delivery (CD) systems, we valued a permissive license and a large developer community, to ensure long-term maintenance of our solution. To keep the hosting of the project flexible, we also did not want to tie us to a specific hosting platform. The following solutions were considered:

- Travis CI[20]
- CircleCI[21]
- GitHub Actions[22]
- Jenkins[23]

Travis CI and CircleCI have a reputation for being more stable than Jenkins because its Plugin system makes the pipelines brittle [Man19]. In "The State of Developer Ecosystem 2023" 51% of developers stated they use Jenkins, with 11% for CircleCI and 9% for Travis CI [Jet23a]. Jenkins is also the only open-source solution of the three.

For our purposes, the open-source licensing and large developer community of Jenkins outweighed the risks of the plugin system, which is why it was chosen.

As our repository was hosted on the *Institute of Microelectronics* "Gitea" instance at the start, the consideration to switch to GitHub Actions, came only when the repository was migrated to GitHub. This would have simplified the setup somewhat, because we could have made use of the direct integration with the GitHub code repository. We decided against this to maintain the flexibility to migrate away from GitHub in the future.

For more information on the CD workflow, see Section 3.5.

3.2.8 Deployment

The main method of deployment is as a Python module through the *Python Package Index* (PyPI)[24]. In addition, we generate standalone Microsoft Windows executable installers for users, who do not have Python installed or lack the knowledge to use PyPI. For this task we use *pynsist*[25].

For more information on the CD workflow, see Section 3.5.

[20] `https://www.travis-ci.com`
[21] `https://circleci.com`
[22] `https://github.com/features/actions`
[23] `https://www.jenkins.io`
[24] `https://pypi.org`
[25] `https://github.com/takluyver/pynsist`

3.3 Implementation

The application can be divided into five general sections:

1. General functions, such as the main entry point into the application and different utility services.
2. The GUI, where all user interface elements are defined in separate classes.
3. The flowchart classes which define the appearance and behavior of the different elements of the flowcharts.
4. The code generator class, which handles the translation from the internal representation of the flowcharts into source code.
5. The debugger classes, which handle debugging in various languages.

For an overview class diagram of the application see Figure 3.1, Figure 3.2 and Figure 3.3.

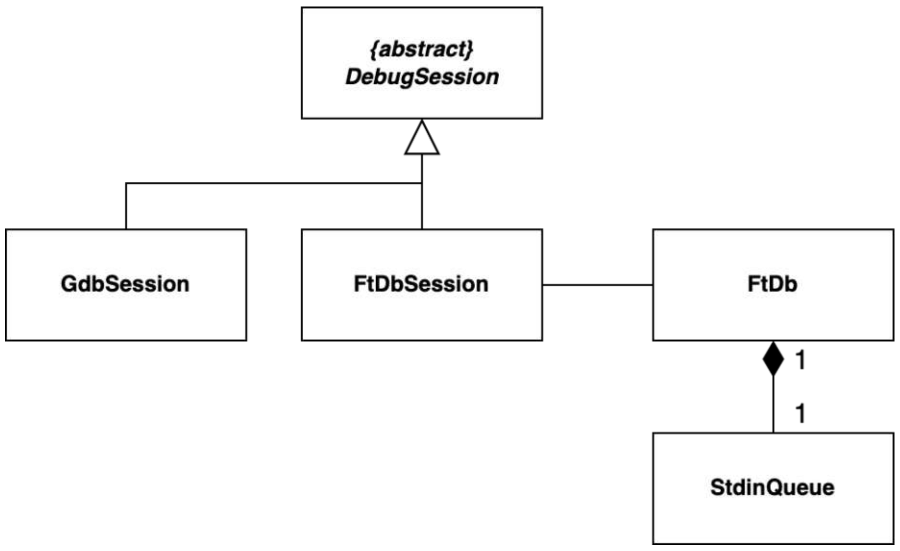


Figure 3.1: Overview class diagram of debugger classes.

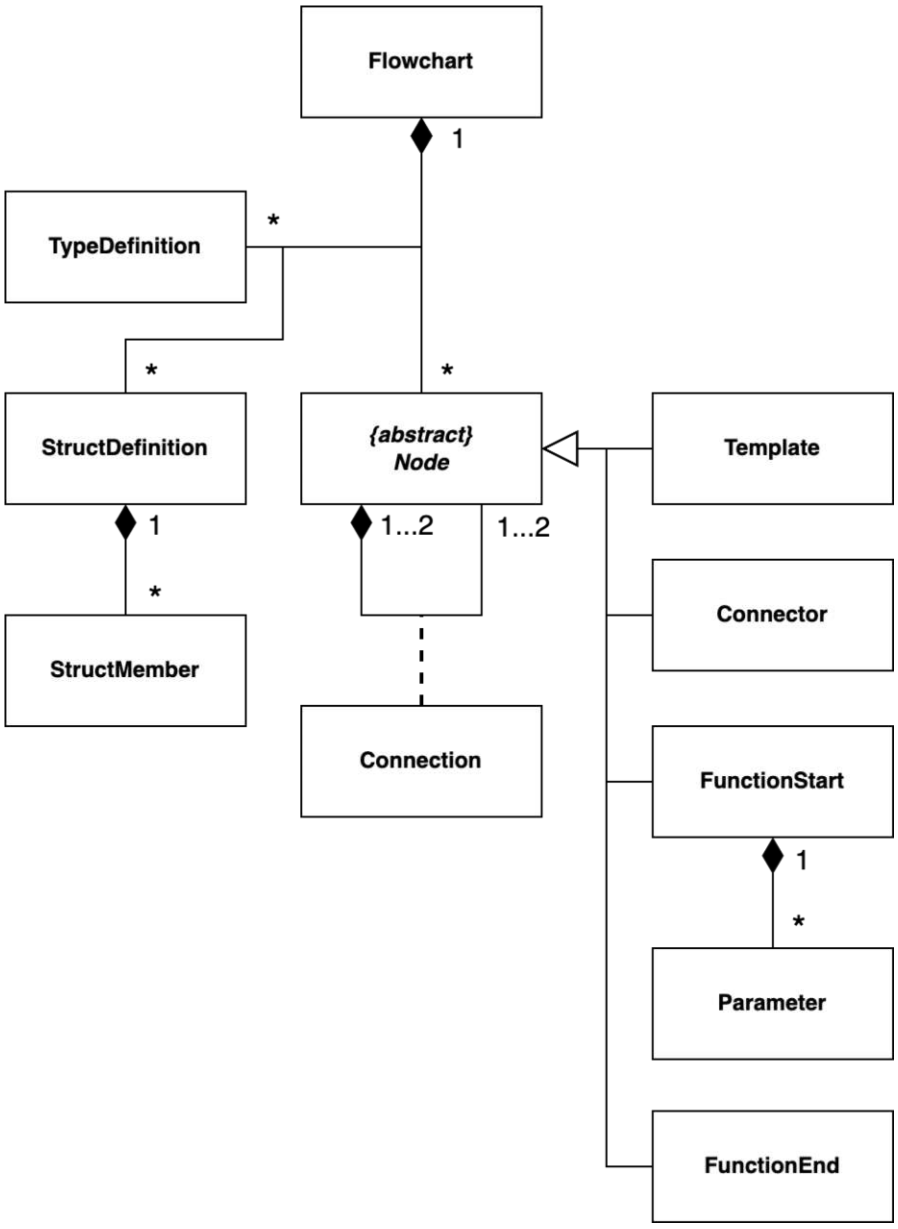


Figure 3.2: Overview class diagram of flowchart classes.

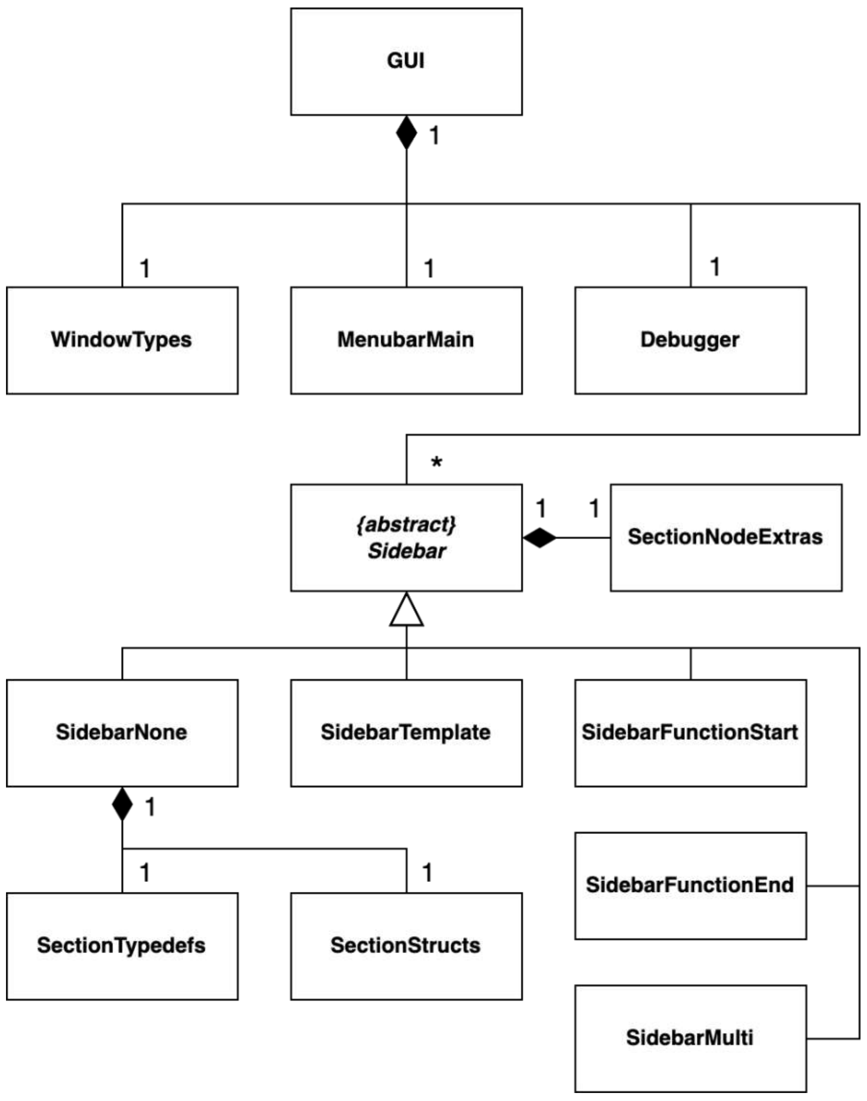


Figure 3.3: Overview class diagram of GUI classes.

In its first iteration FlowTutor was intended for a lecture on introductory C programming and some features and implementation details still reflect this. In the latest version, FlowTutor was partially rewritten, to support custom language templates, with which users can define custom language definitions, see also Section 3.3.3.

3.3.1 General

In FlowTutor various service classes are injected through Dependency Injection (DI). The fundamental concept behind Dependency Injection involves using a separate object, in the implementation employed here called a *container*, which is responsible for providing a concrete implementation to a constructor parameter, which is defined as an interface. This form of DI is also called *Constructor Injection* [Fow08].

The following service classes are instantiated as Singletons[GHJV94] in the `Container` class:

- `LanguageService`
  This service facilitates using programming languages defined in the templates folder.
- `ModalService`
  A service to handle the showing of modal dialogs throughout the GUI.
- `SettingsService`
  A Service for storing and retrieving settings between executions of FlowTutor.
- `UtilService`
  Various helper methods, e.g., to get file system paths.

3.3.2 GUI

The `GUI` class is instantiated once at the startup of the application and contains the program state in its fields.

Upon startup of the application, the users are greeted with a welcome modal, see Figure 3.4, where they can either create a new project, by selecting the programming language, or opening an existing project. For convenience, the most recently opened projects are listed.

After the project selection, the users are presented with the main interface, as shown in Figure 3.8.

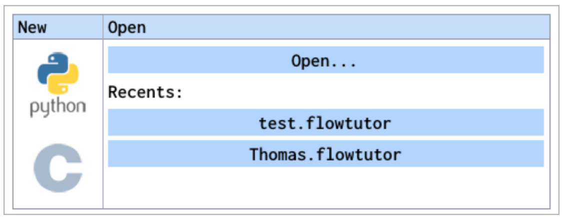
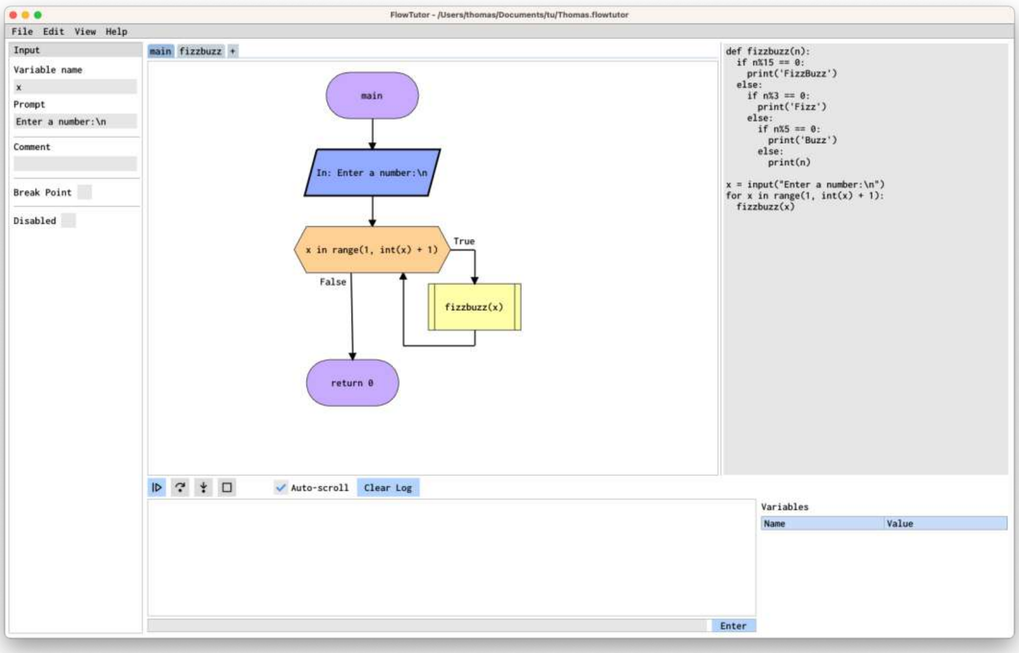


Figure 3.4: The welcome modal window of FlowTutor.

The interface is separated into different sections:

Top menu bar

The menu bar at the top (see Figure 3.5) contains general file commands such as creating a new program, saving and loading projects, etc. In the *View* menu, FlowTutor can be switched into a dark mode, which can be seen in Figure 3.6.



Input

Variable name

x

Prompt

Enter a number:\n

Comment

Break Point

Disabled

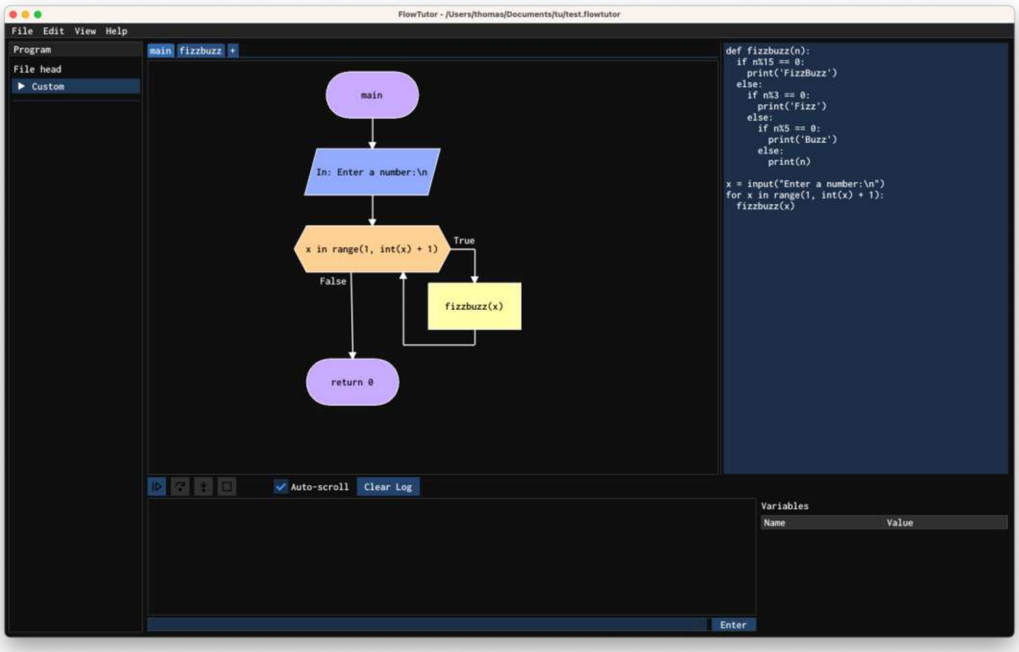
```
def fizzbuzz(n):
  if n%15 == 0:
    print('FizzBuzz')
  else:
    if n%3 == 0:
      print('Fizz')
    else:
      if n%5 == 0:
        print('Buzz')
      else:
        print(n)

x = input("Enter a number:\n")
for x in range(1, int(x) + 1):
  fizzbuzz(x)
```

Variables

Name	Value

Figure 3.5: The main view of FlowTutor.



```
def fizzbuzz(n):
  if n%15 == 0:
    print('FizzBuzz')
  else:
    if n%3 == 0:
      print('Fizz')
    else:
      if n%5 == 0:
        print('Buzz')
      else:
        print(n)

x = input("Enter a number:\n")
for x in range(1, int(x) + 1):
  fizzbuzz(x)
```

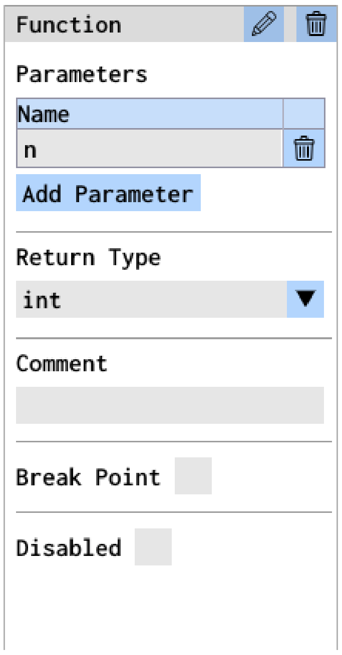
Variables

Name	Value

Figure 3.6: FlowTutor in dark mode.

Left sidebar

The sidebar on the left changes depending on the selected node in the drawing area. As an example, Figure 3.7 shows the sidebar, when a function definition is selected. In this case, the user can add, remove, and edit parameters and select a return type of the function. In Figure 3.5 an *input* node is selected, which results in different options being shown in the sidebar.



Function

Parameters

Name	
n	🗑

Add Parameter

Return Type

int ▼

Comment

Break Point

Disabled

Figure 3.7: The sidebar for a function definition in FlowTutor.

For all node types, there are three default options, in the lower part of the sidebar:

- A text field for a code comment
- A checkbox which toggles a debugging breakpoint on this node
- A checkbox which disables the node and effectively comments it out in the source code

If no node is selected, the sidebar shows project-wide parameters. These differ, depending on the programming language of the project. Figure 3.8 shows that in Python it is currently only possible to insert custom code at the beginning of the source code file. In contrast, Figure 3.9 shows the options for a C project, which include importable headers, custom definitions, and a button to access the type definition window.

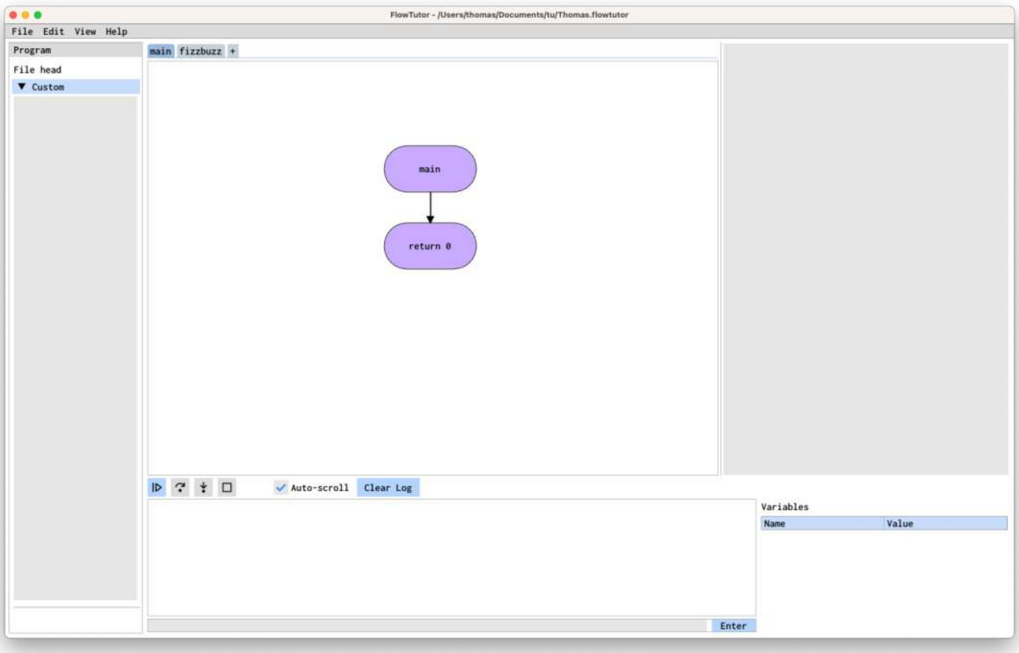
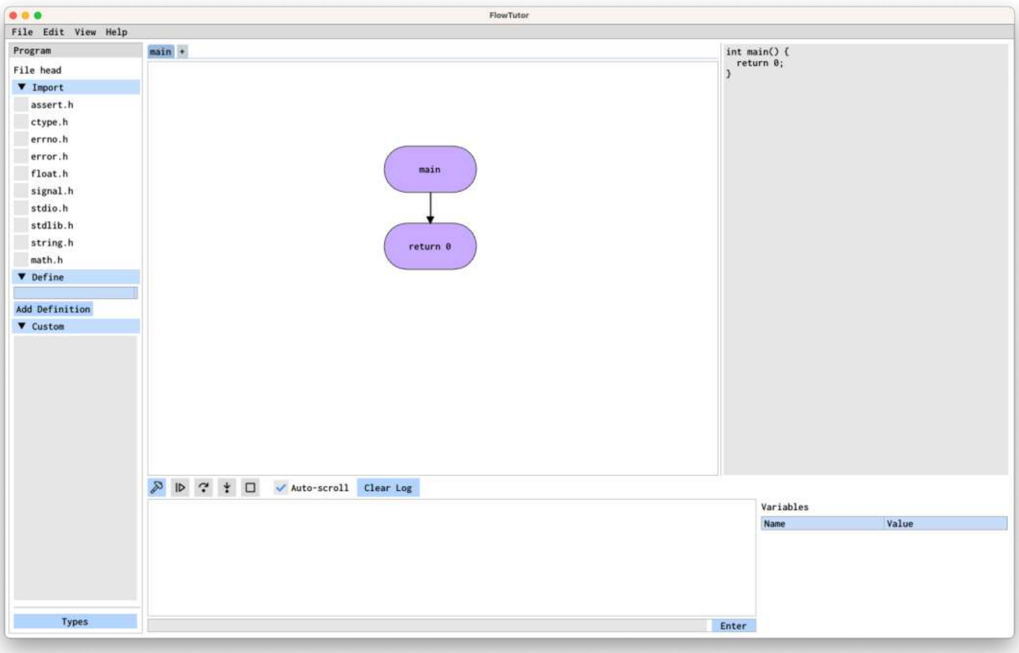


Figure 3.8: An empty FlowTutor project in Python.



```
int main() {
  return 0;
}
```

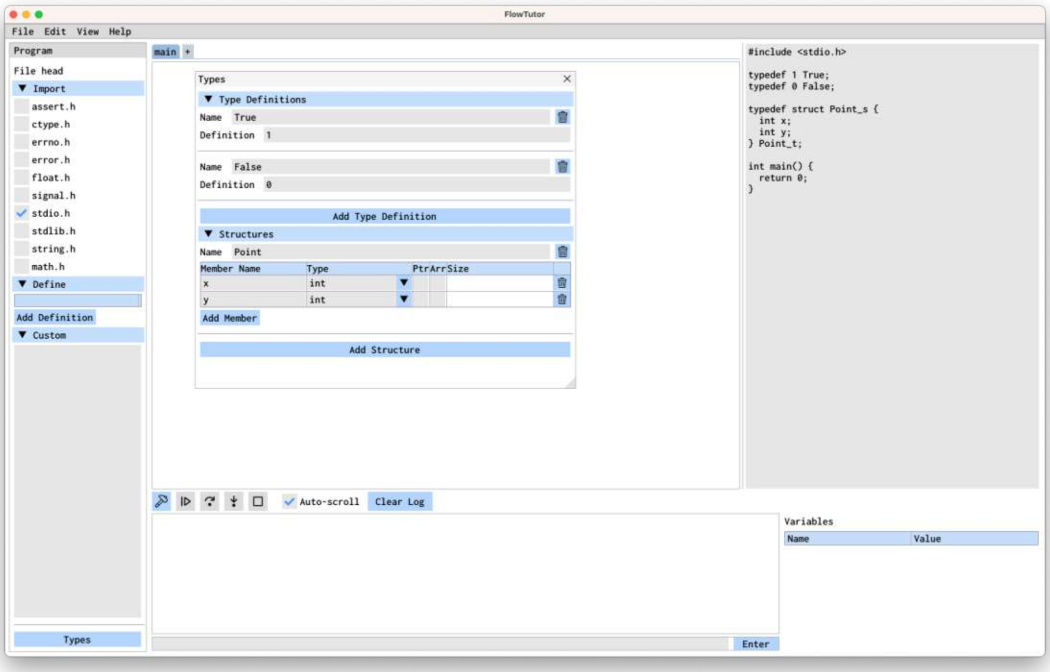
Figure 3.9: An empty FlowTutor project in C.

Type definition window

This optional window can only be accessed for C projects. Here, the users can declare type definitions as well as structure definitions (see Figure 3.10).

Types are defined by entering the name of the new type and the existing type to which it should refer.

Structures are defined by entering the name and members of the new structure, with the members in turn being defined by their name, type, and whether they are pointers or arrays. In the case of array members, the size must be defined as well.



Type Definitions

Name True
Definition 1

Name False
Definition 0

Structures

Name Point

Member Name	Type	Ptr	Arr	Size
x	int			
y	int			

```
#include <stdio.h>

typedef 1 True;
typedef 0 False;

typedef struct Point_s {
  int x;
  int y;
} Point_t;

int main() {
  return 0;
}
```

Figure 3.10: Various C type-definitions and structure definitions in a FlowTutor project.

Center drawing area

In the center of the screen is the main area, where the flowcharts are created. Directly above it, is a tabbed interface, where users can switch between different flowcharts, representing function definitions. Initially, a main function is created for every project and new functions can be added with the “+” button. For more information on the flowcharts see Section 3.3.3.

Right source code area

In the right text area, the source code is displayed in real-time, meaning that a change in the flowchart is immediately reflected in the source code. As an example see an implementation of the FizzBuzz problem in Figure 3.11.

Bottom debugger

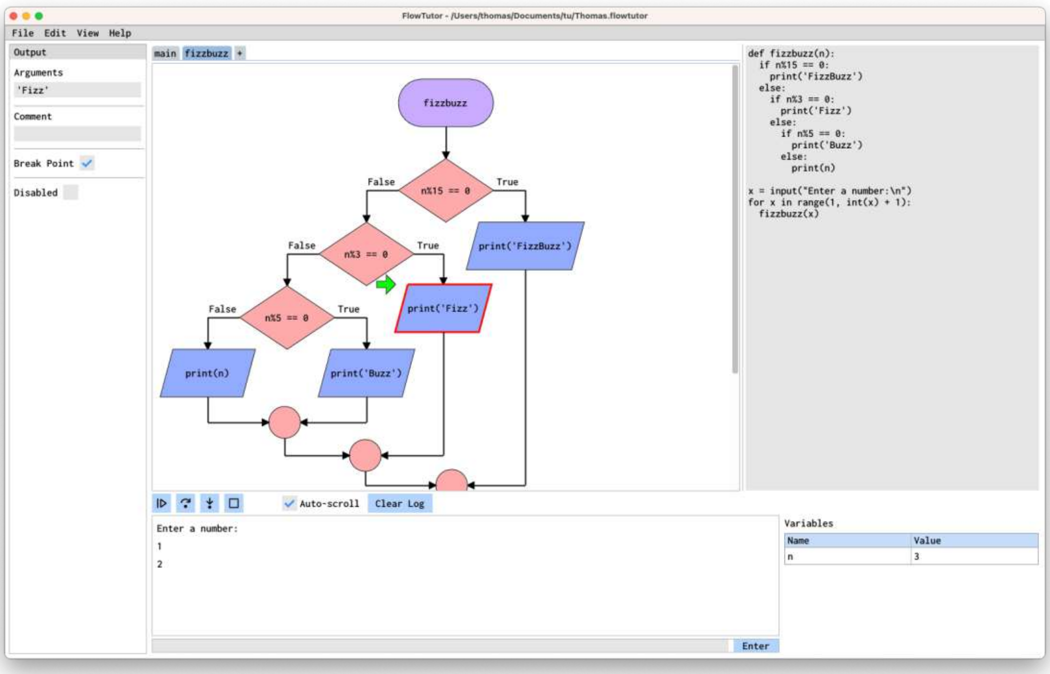
In the bottom area, below the drawing area and source code area, is the debugger. See also Figure 3.11.

At the top of the debugger are the control buttons:

- Compiling the program (only shown for C projects).
- Running the program.
- Stepping over the current statement.
- Stepping into the current statement.
- Stopping execution.

Below these controls is a console which provides an interface for text input and output. In the Windows version of FlowTutor, the console only shows compiler messages, such as warnings and errors for C programs. This is due to the fact that Windows does not support TTY interfaces for the built-in console. For more information on the implementation of the debugger see section 3.3.6.

To the right of the console is a table which shows variable assignments during execution.



```
def fizzbuzz(n):
  if n%15 == 0:
    print('FizzBuzz')
  else:
    if n%3 == 0:
      print('Fizz')
    else:
      if n%5 == 0:
        print('Buzz')
      else:
        print(n)

x = input("Enter a number:\n")
for x in range(1, int(x) + 1):
  fizzbuzz(x)
```

```
Enter a number:
1
2
```

Variables

Name	Value
n	3

Figure 3.11: The FlowTutor debugger.

3.3.3 Flowcharts

A *FlowTutor* project is a list of `Flowchart` objects, each representing a function and associated metadata.

Each `Flowchart` object contains a root `FunctionStart` object which is derived from the abstract base class `Node`. The `FunctionStart` node contains a single `Connection`, which connects it to its predecessor. For new flowcharts, this is always a `FunctionEnd` node, see Figure 3.8.

In the first iteration of *FlowTutor*, the only supported programming language was C. Each node type was represented through its own class derived from `Node`. The following C constructs were available:

- Variable declaration
- Variable assignment
- Conditional
- For-loop
- While-loop

- Do-while-Loop
- Input
- Output
- Function call
- Code snippets

A follow-up feature enhancement introduced the `Template` node type. With this node type, the users have the ability to create custom nodes through JSON-formatted definition files. The source code these template nodes produce is also defined in the definition file. The `Template` node type was iterated upon until it could replace all the hard-coded derived node classes. The template system is described in detail in Section 3.3.4.

Depending on the type of node, it has a connection to one or multiple child nodes. The only exception is the `FunctionEnd` node, which has no outgoing connection. The users insert new nodes, by clicking at the start of the connection, where they plan to insert the new node. The node is inserted with the appropriate connections and existing nodes are shifted to make space and avoid overlapping nodes in the drawing area.

The users can select nodes by clicking on them, or by dragging a selection fence over them. The node-specific settings appear in the sidebar on the left.

All changes are translated in real time to their corresponding source-code representation. See also Section 3.3.5.

3.3.4 Node Templates

To create a flexible system which can support arbitrary programming languages and constructs, we decided to base the available programming languages on user-definable templates. These nodes are configured with JSON files and each node type has a corresponding configuration. Every available language has its own directory in the *FlowTutor* directory structure.

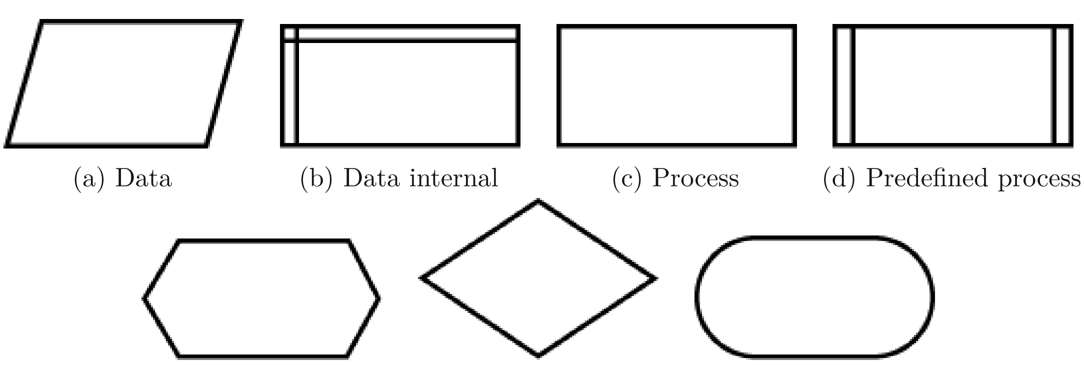
We wanted to avoid reinventing the wheel with our source code template syntax, which is why used an existing solution with Jinja2[26].

The configuration files contain the following properties:

- `label`
  The name of the node shown to the user (e.g., on button labels).
- `node_label` (optional)
  The text that is displayed on the drawn node in the flowchart. Can be a Jinja template string. See also the body property.

[26]`https://jinja.palletsprojects.com`

- `shape_id`
  The shape of the node. Can be one of the following:
  - `data` (see Figure 3.12a)
  - `data_internal` (see Figure 3.12b)
  - `process` (see Figure 3.12c)
  - `predefined_process` (see Figure 3.12d)
  - `preparation` (see Figure 3.12e)
  - `decision` (see Figure 3.12f)
  - `terminator` (see Figure 3.12g)



(a) Data (b) Data internal (c) Process (d) Predefined process

(e) Preparation (f) Decision (g) Terminator

Figure 3.12: Node shapes based on ISO 5807 [iso85].

- `control_flow` (optional)
  Defines how the connections work for this node. There are four possibilities for this property:
  - No value, resulting in a single input and output.
  - `loop` (see Figure 3.13a)
  - `post-loop` (see Figure 3.13b)
  - `decision` (see Figure 3.13c)

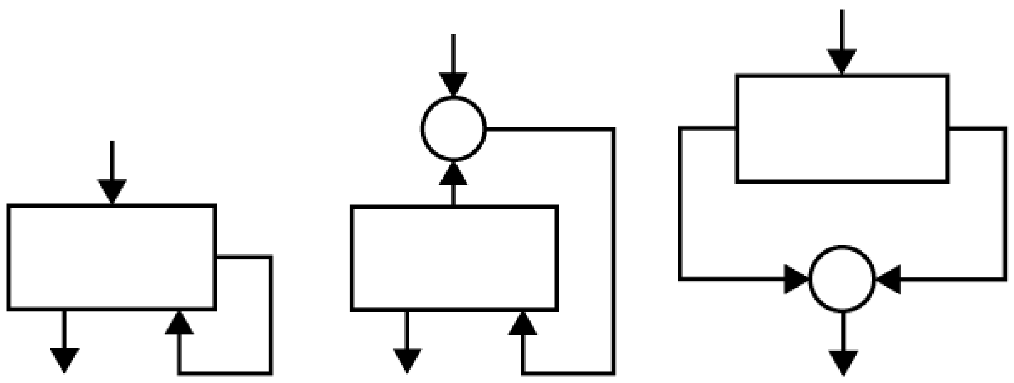
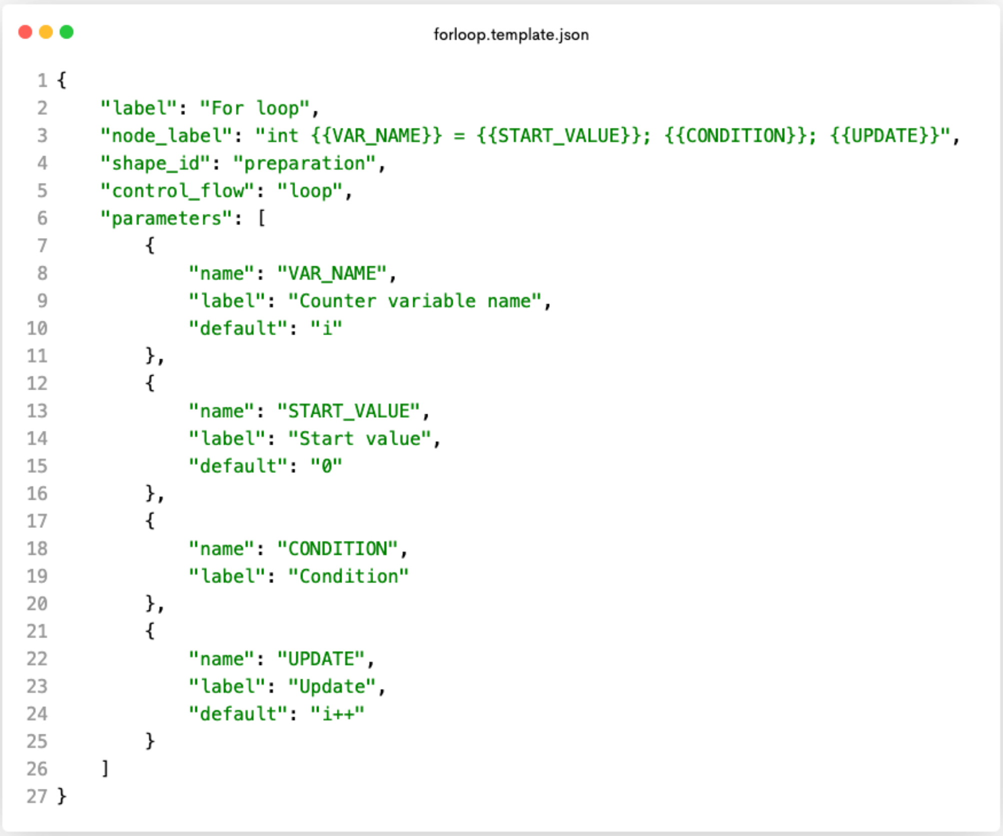


Figure 3.13: Control-flow variations.

- `color` (optional)
  The color of the node, defined by its RGB values.
- `parameters`
  The user-facing parameters which can be changed when using the node. A parameter has the following properties:
  - `name`
    The name that is referenced in the template body to insert the argument.
  - `label`
    The user-facing label of the parameter.
  - `default`
    The default value of the parameter.
  - `options`
    A predefined list of values to which the the parameter is constrained. The user is presented with a drop-down list.
  - `visible`
    An expression in Python syntax which can reference other parameters to conditionally hide this parameter.
- `body` (optional)
  A string which is inserted in the source code. If this value is omitted, a Jinja template file of the same name as the configuration must be present (e.g., for `while-loop.template.json` there must be a file called `while-loop.jinja`)

For an example of a template file see Figure 3.14.


forloop.template.json

```
{
    "label": "For loop",
    "node_label": "int {{VAR_NAME}} = {{START_VALUE}}; {{CONDITION}}; {{UPDATE}}",
    "shape_id": "preparation",
    "control_flow": "loop",
    "parameters": [
        {
            "name": "VAR_NAME",
            "label": "Counter variable name",
            "default": "i"
        },
        {
            "name": "START_VALUE",
            "label": "Start value",
            "default": "0"
        },
        {
            "name": "CONDITION",
            "label": "Condition"
        },
        {
            "name": "UPDATE",
            "label": "Update",
            "default": "i++"
        }
    ]
}
```

Figure 3.14: Template for a C “For”-loop node.

We have predefined templates for the C and Python programming languages, which are to be adapted and extended in the future:

C Nodes

- Declaration (Figure 3.15)
  A variable declaration; used to create variables of all available basic types, as well as pointers and arrays of these types. Variables can be declared as static and have a default value.
- Assignment (Figure 3.16)
  Variable assignment; used to assign values to variables.
- Call (Figure 3.17)
  Primarily used to make function calls, but can be used to execute any arbitrary expression.
- Conditional (Figure 3.18)
  Represents a conditional statement which divides the execution path into two, based on a condition expression. If the "else" branch is left empty, the corresponding source code does not include it.
- While-Loop (Figure 3.19)
  A "while" loop which is executed based on a condition expression.
- Do-While-Loop (Figure 3.20)
  A post-test "do-while" loop which is executed based on a condition expression.
- For-Loop (Figure 3.21)
  A "for" loop which is executed based on a condition expression, where initialization, evaluation, and iteration are part of the syntactic construct.
- Function Start (Figure 3.22)
  This node represents a function declaration and definition. Parameters are defined by their type and name. Every flowchart in *FlowTutor* is a function, so a "Function Start" node is always the root node of every flowchart. The function body gets defined by adding more nodes after this one.
- Function End (Figure 3.23)
  Used to terminate a flowchart and to define a return value for the current function.
- Input (Figure 3.24)
  Reads a value from the end-user, using the built-in `scanf` function. Its parameters are the name of the variable to which the data is being written, a template specifier, which handles the type of the input data, and a user-facing prompt message. Optionally the node can generate a variable declaration if it was not declared previously.

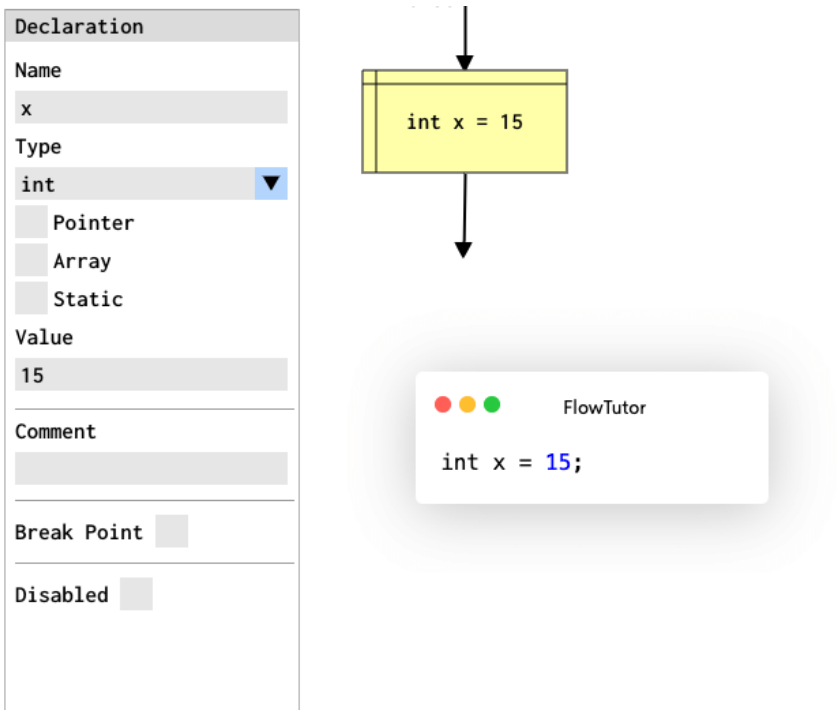
- Output (Figure 3.25)
  Prints text to the console using the built-in `printf` function. Its parameters are a template expression and corresponding arguments.

- Open File (Figure 3.26)
  Declares a variable for a pointer to a file on disk and uses `fopen` to open it. The parameters are the path on disk and the mode in which the file should be opened.

- Write File (Figure 3.27)
  Writes a value to a file on disk. The parameters are the name of a pointer variable of a previously opened file and the value which is to be written.

- Close File (Figure 3.28)
  Closes a previously opened file using its pointer variable.

- Snippet (Figure 3.29)
  Arbitrary multi-line code used to insert code which is not otherwise available as a dedicated node type.



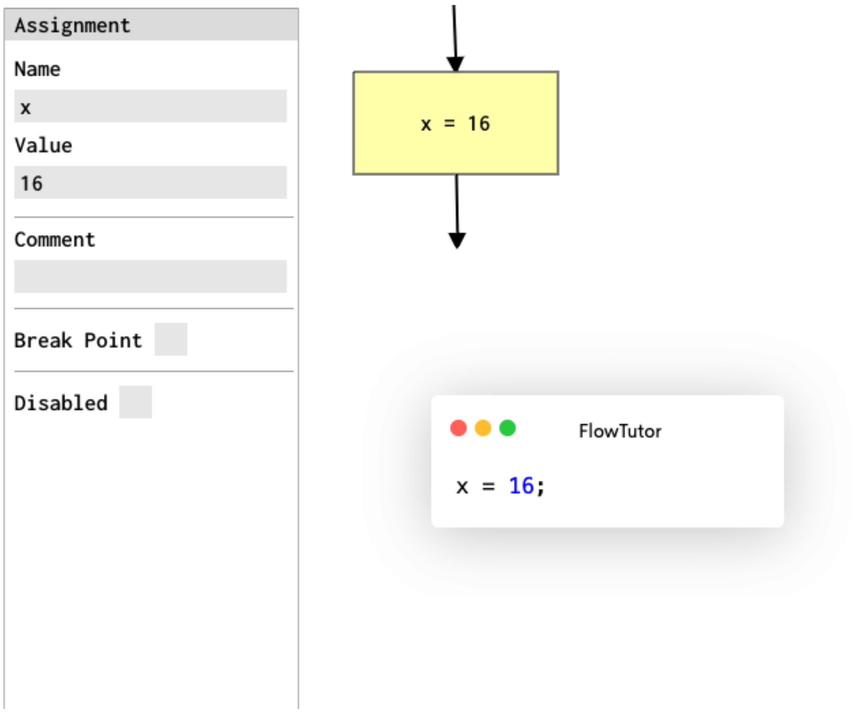
Declaration
Name: x
Type: int
Pointer
Array
Static
Value: 15
Comment:
Break Point
Disabled

```
int x = 15
```

FlowTutor

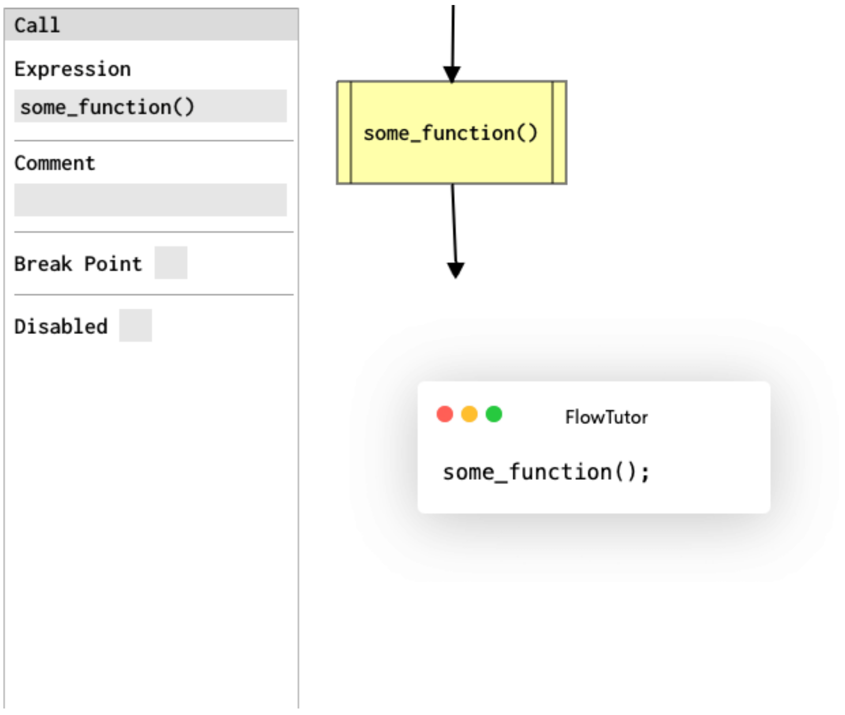
```
int x = 15;
```

Figure 3.15: C “Declaration” node and resulting source code.



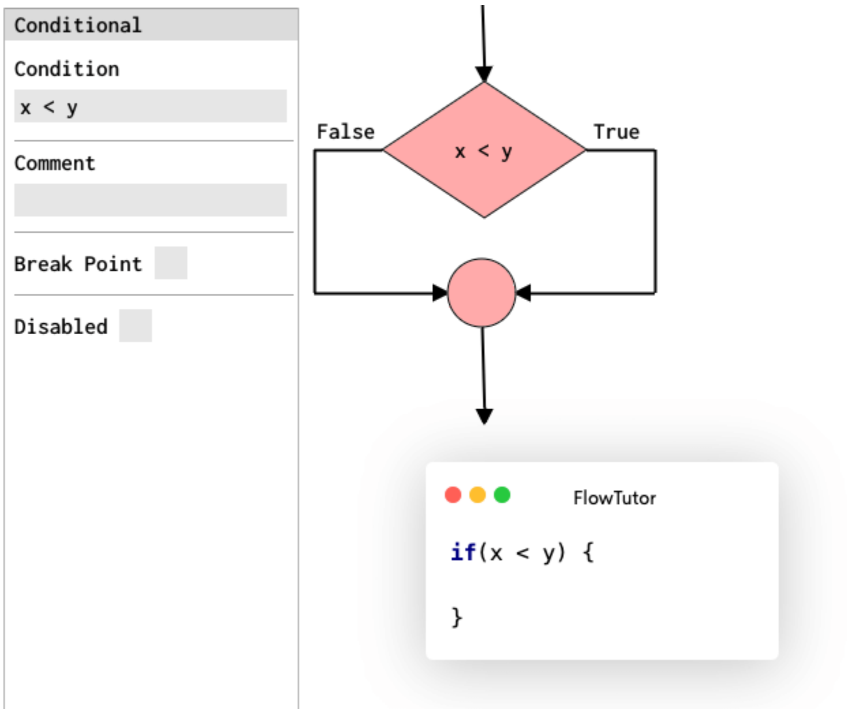
```
x = 16;
```

Figure 3.16: C “Assignment” node and resulting source code.



```
some_function();
```

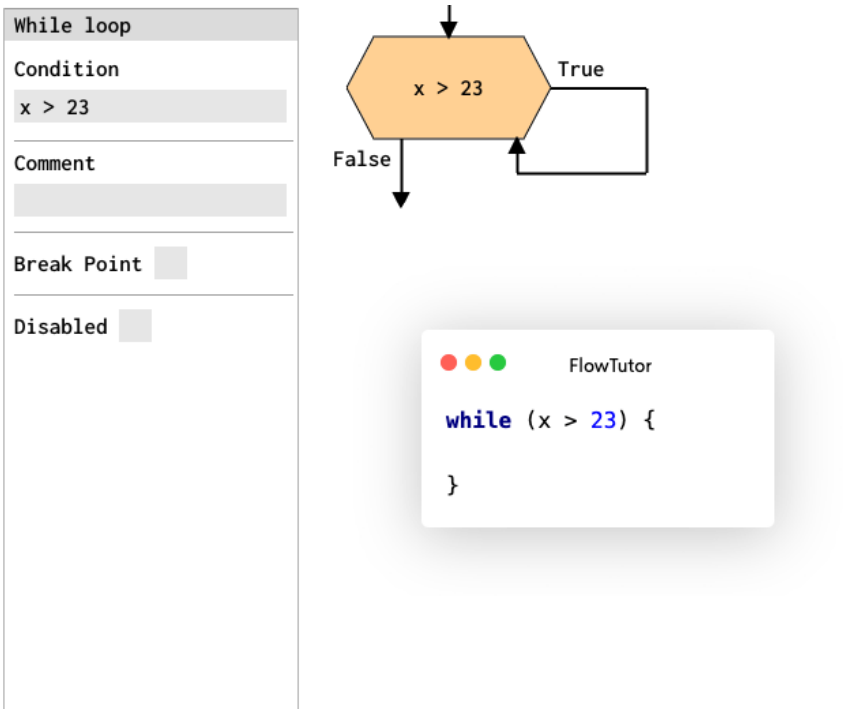
Figure 3.17: C “Call” node and resulting source code.



```
if(x < y) {

}
```

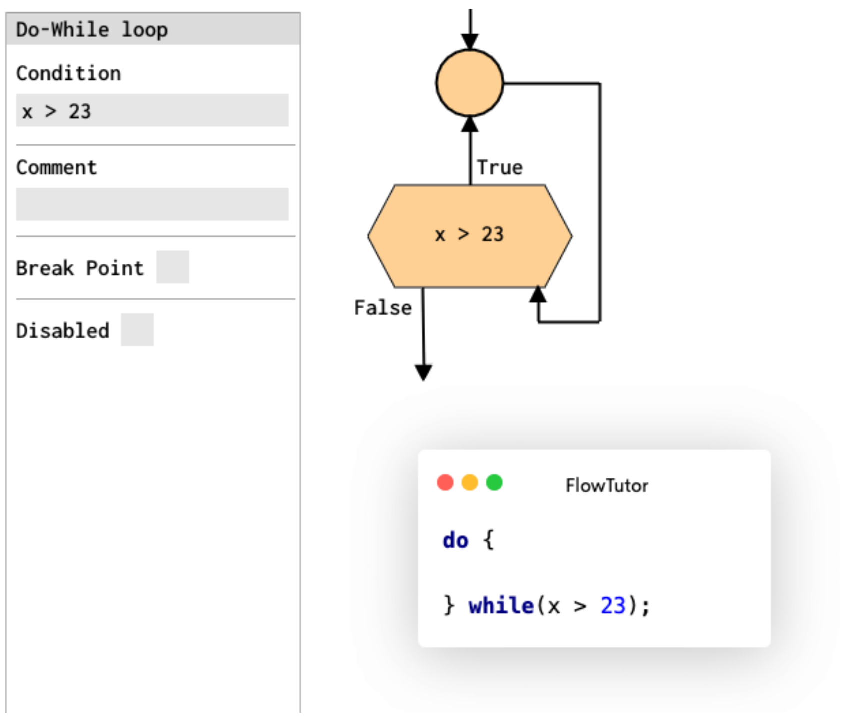
Figure 3.18: C "Conditional" node and resulting source code.



```
while (x > 23) {

}
```

Figure 3.19: C "While"-loop node and resulting source code.



Do-While loop

Condition

x > 23

Comment

Break Point

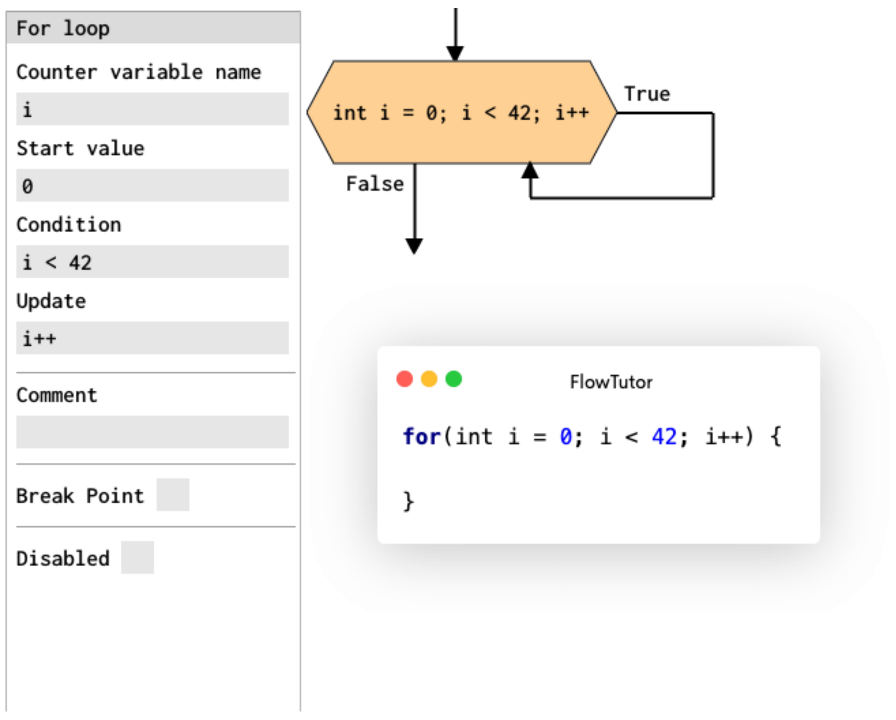
Disabled

FlowTutor

```c
do {

} while(x > 23);
```

Figure 3.20: C “Do-While”-loop node and resulting source code.



For loop

Counter variable name

i

Start value

0

Condition

i < 42

Update

i++

Comment

Break Point

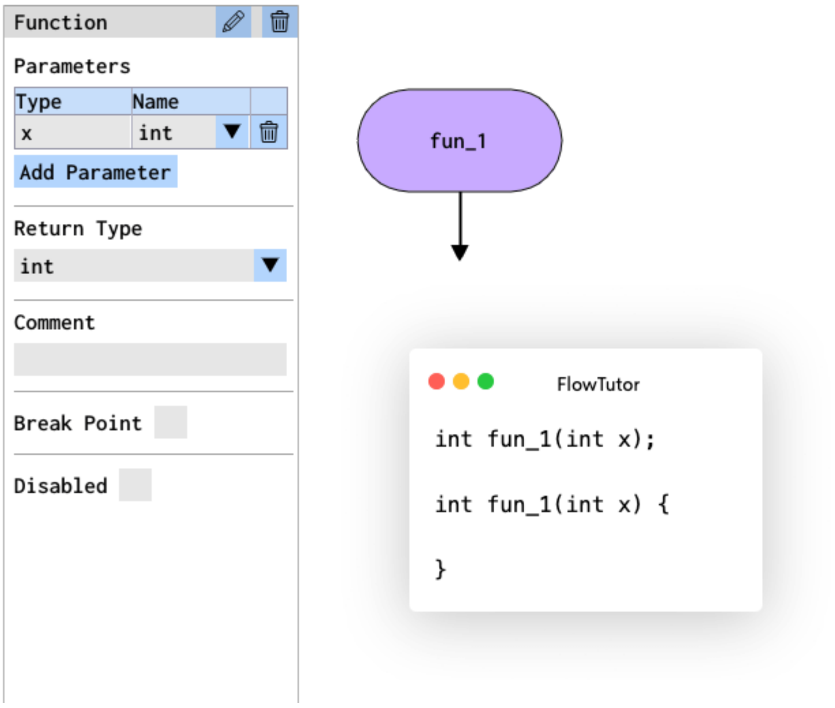
Disabled

FlowTutor

```c
for(int i = 0; i < 42; i++) {

}
```

Figure 3.21: C “For”-loop node and resulting source code.



Parameters: x, int

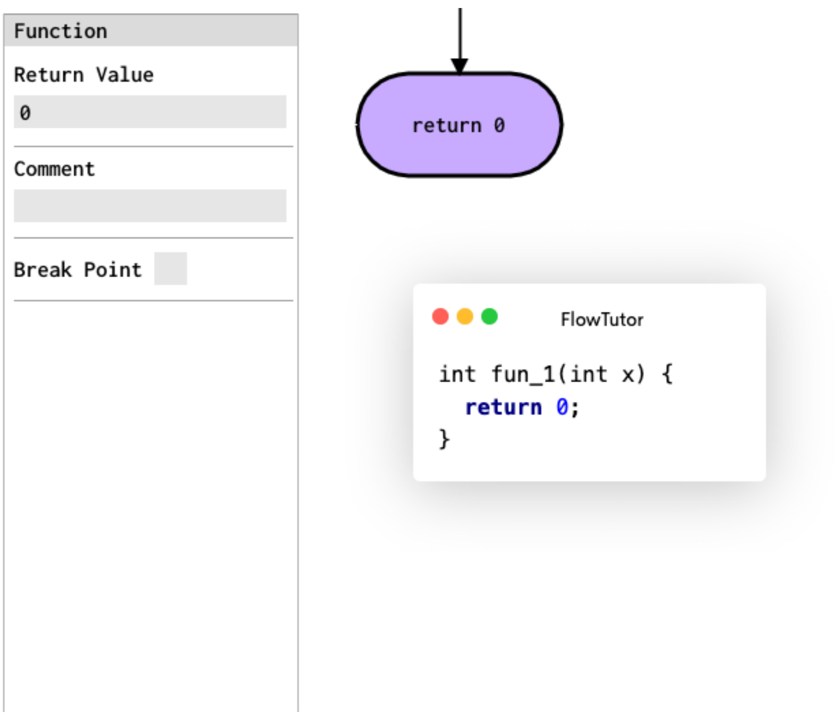
Return Type: int

```
int fun_1(int x);

int fun_1(int x) {

}
```

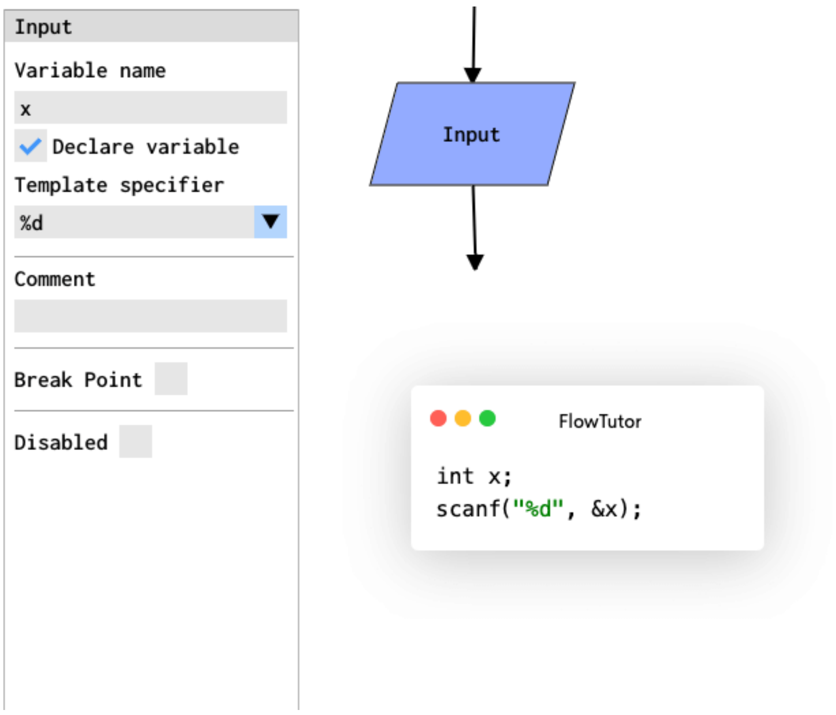
Figure 3.22: C “Function Start” node and resulting source code.



Return Value: 0

```
int fun_1(int x) {
  return 0;
}
```

Figure 3.23: C “Function End” node and resulting source code.



Input

Variable name

x

✓ Declare variable

Template specifier

%d

Comment

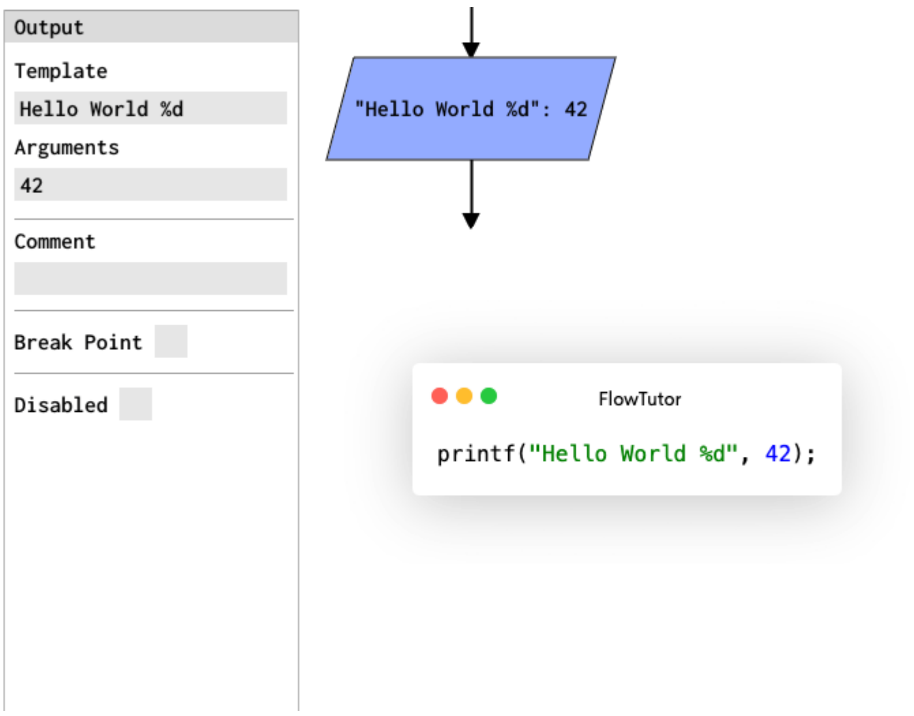
Break Point

Disabled

```
FlowTutor

int x;
scanf("%d", &x);
```

Figure 3.24: C "Input" node and resulting source code.



Output

Template

Hello World %d

Arguments

42

Comment

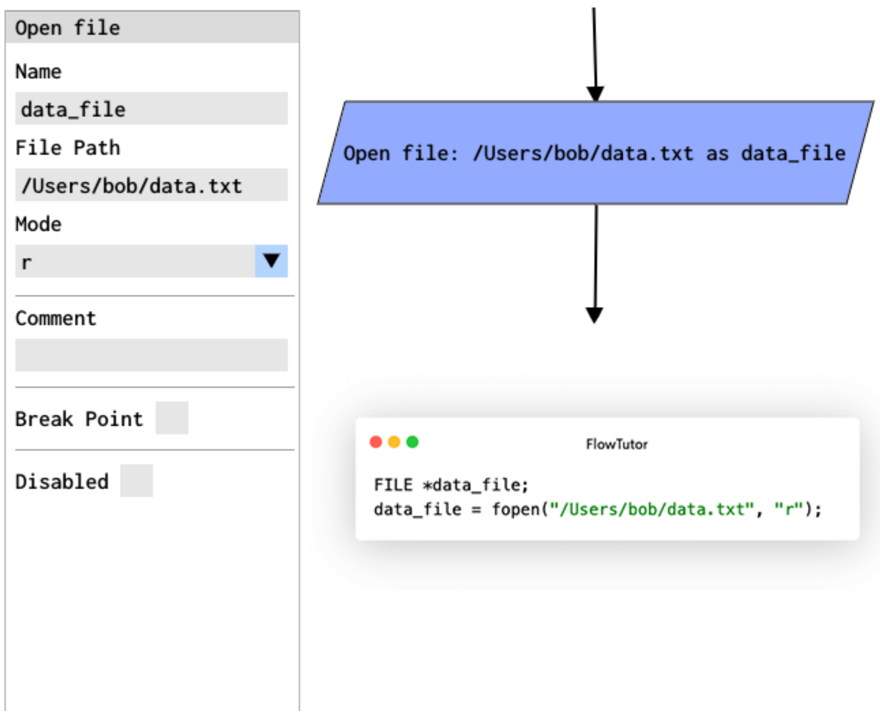
Break Point

Disabled

```
FlowTutor

printf("Hello World %d", 42);
```

Figure 3.25: C "Output" node and resulting source code.

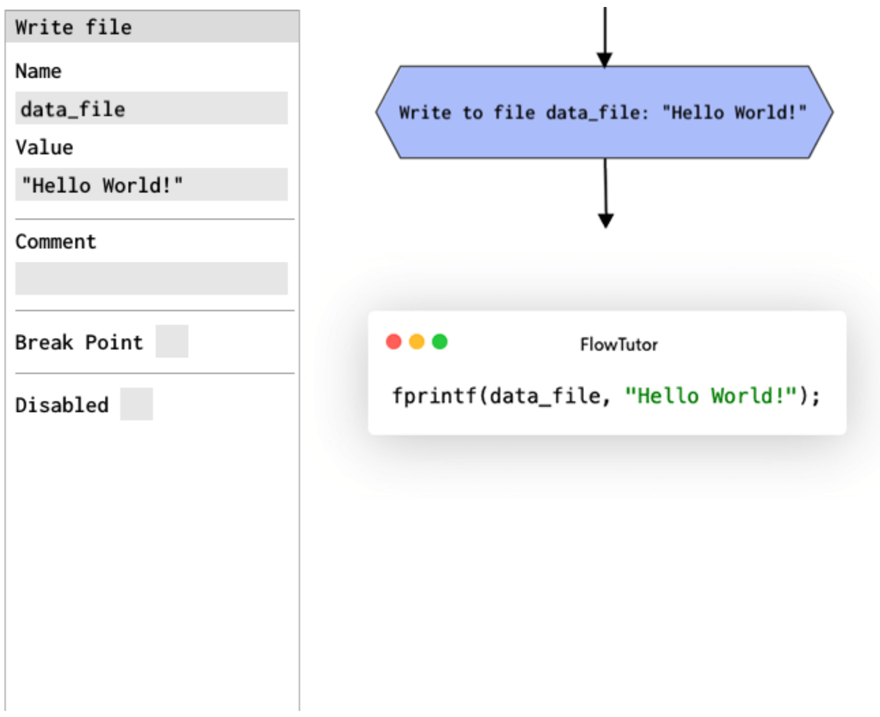


Name	data_file
File Path	/Users/bob/data.txt
Mode	r
Comment	

```
Open file: /Users/bob/data.txt as data_file
```

```c
FILE *data_file;
data_file = fopen("/Users/bob/data.txt", "r");
```

Figure 3.26: C “Open File” node and resulting source code.

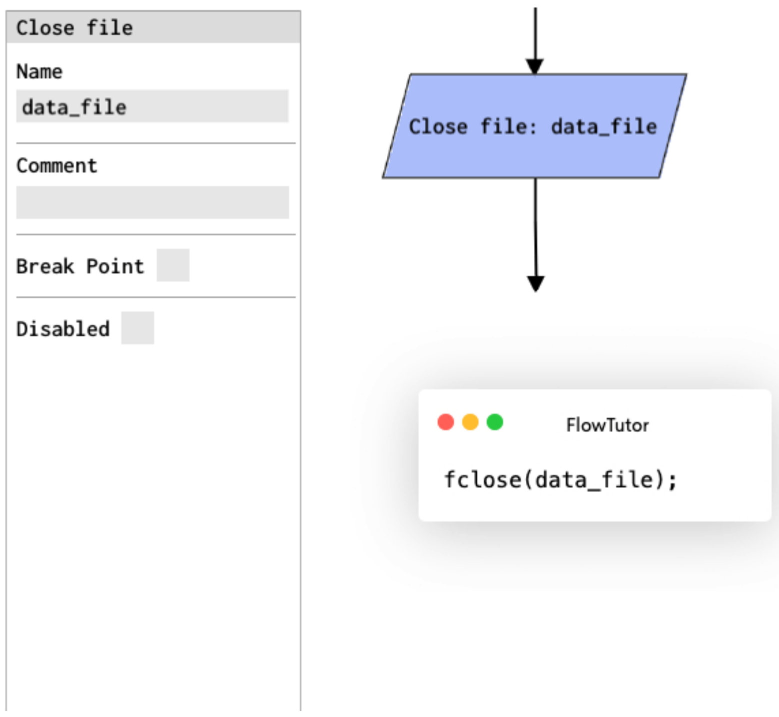


Name	data_file
Value	"Hello World!"
Comment	

```
Write to file data_file: "Hello World!"
```

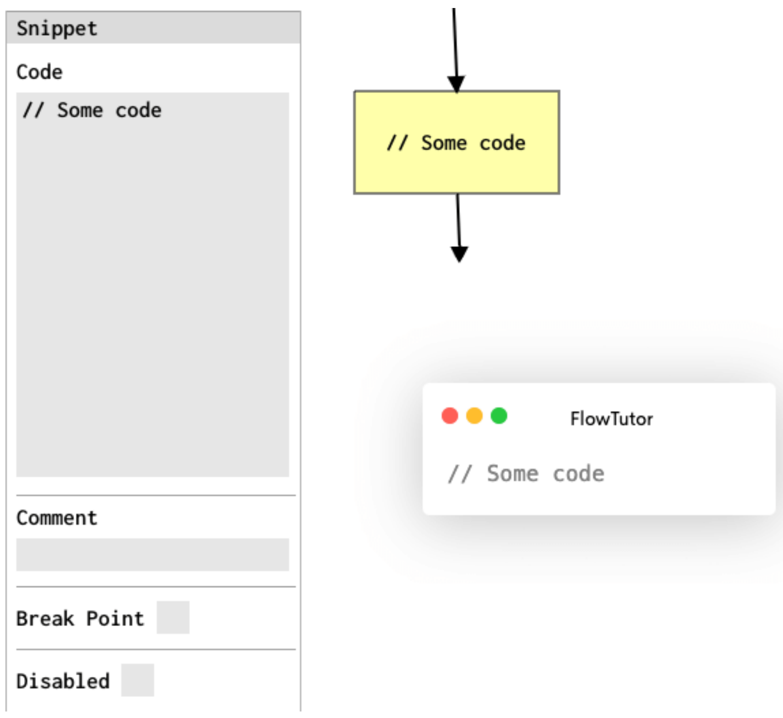
```c
fprintf(data_file, "Hello World!");
```

Figure 3.27: C “Write File” node and resulting source code.



```
fclose(data_file);
```

Figure 3.28: C “Close file” node and resulting source code.

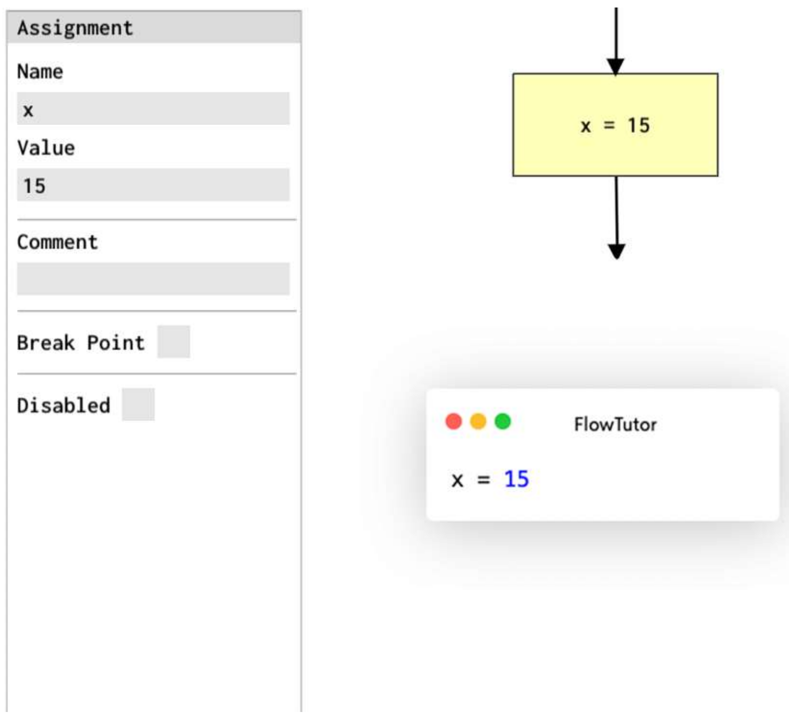


```
// Some code
```

Figure 3.29: C “Snippet” node and resulting source code.

Python Nodes

- Assignment (Figure 3.30)
  Assigns a value to a variable.
- Call (Figure 3.31)
  Primarily used to make function calls, but can be used to execute any arbitrary expression.
- Conditional (Figure 3.32)
  Represents a conditional statement which divides the execution path into two, based on a condition expression. If the "else" branch is left empty, the corresponding source code does not include it.
- While-Loop (Figure 3.33)
  A "while" loop which is executed based on a condition expression.
- For-Loop (Figure 3.34)
  A "for" loop which is executed for every member of an iterable object.
- Function Start (Figure 3.35)
  This node represents a function definition. Parameters are defined by their name. Every flowchart in *FlowTutor* is a function, therefore a "Function Start" node is always the root node of every flowchart. The function body is defined by adding more nodes after this one.
- Function End (Figure 3.36)
  Used to terminate a flowchart and to define a return value for the current function.
- Input (Figure 3.37)
  Reads a value from the end-user, using the built-in `input` function. Its parameters are the name of the variable, to which the data is being written and a user-facing prompt message.
- Output (Figure 3.38)
  Prints text to the console, using the built-in `print` function. Its parameters are a template expression and corresponding arguments.
- Snippet (Figure 3.39)
  Arbitrary multi-line code used to insert code which is not otherwise available as a dedicated node type.



Assignment

Name

x

Value

15

Comment

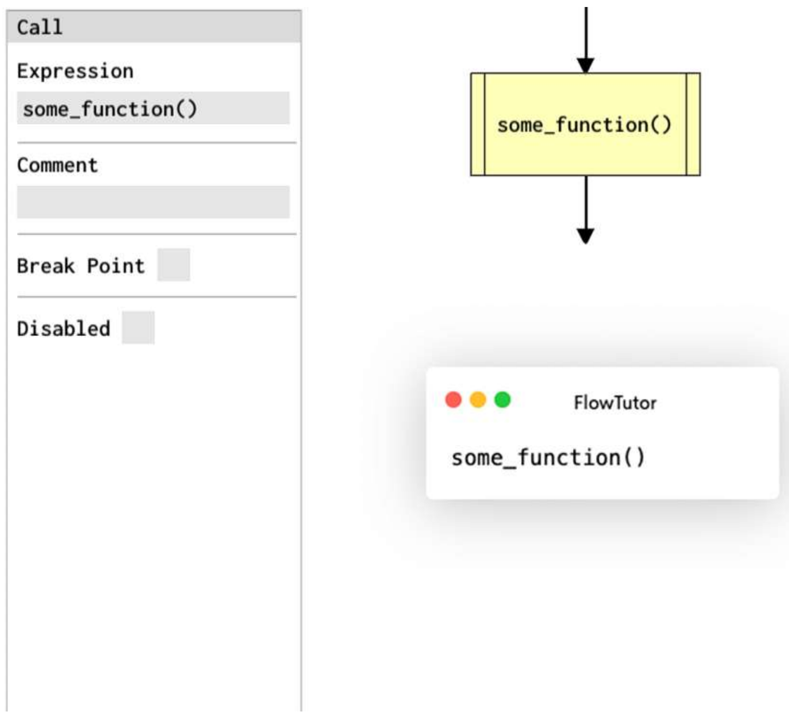
Break Point

Disabled

```
x = 15
```

```
x = 15
```

Figure 3.30: Python "Assignment" node and resulting source code.



Call

Expression

some_function()

Comment

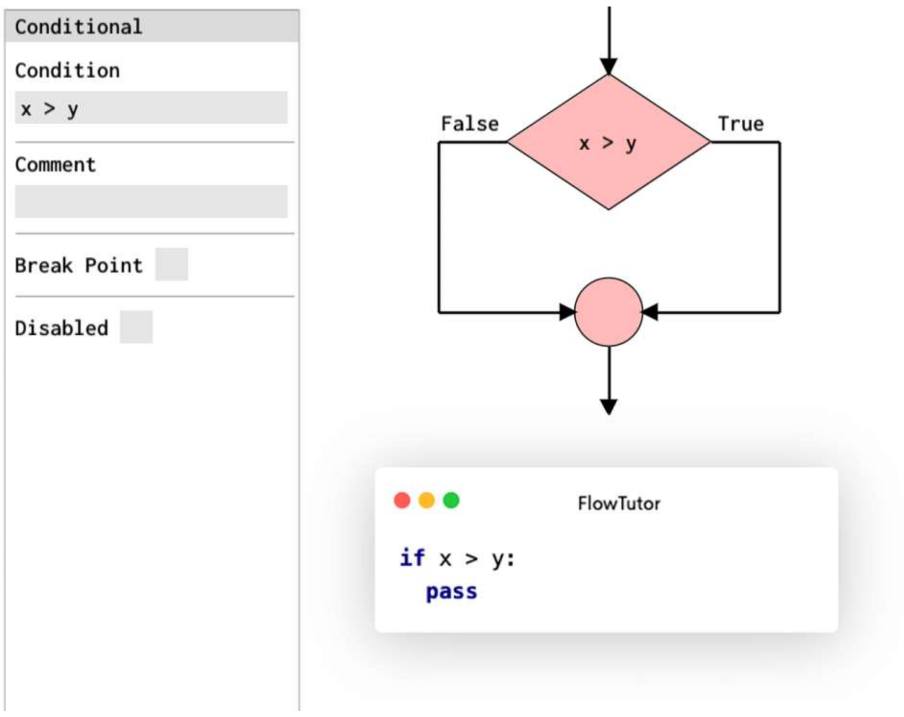
Break Point

Disabled

```
some_function()
```

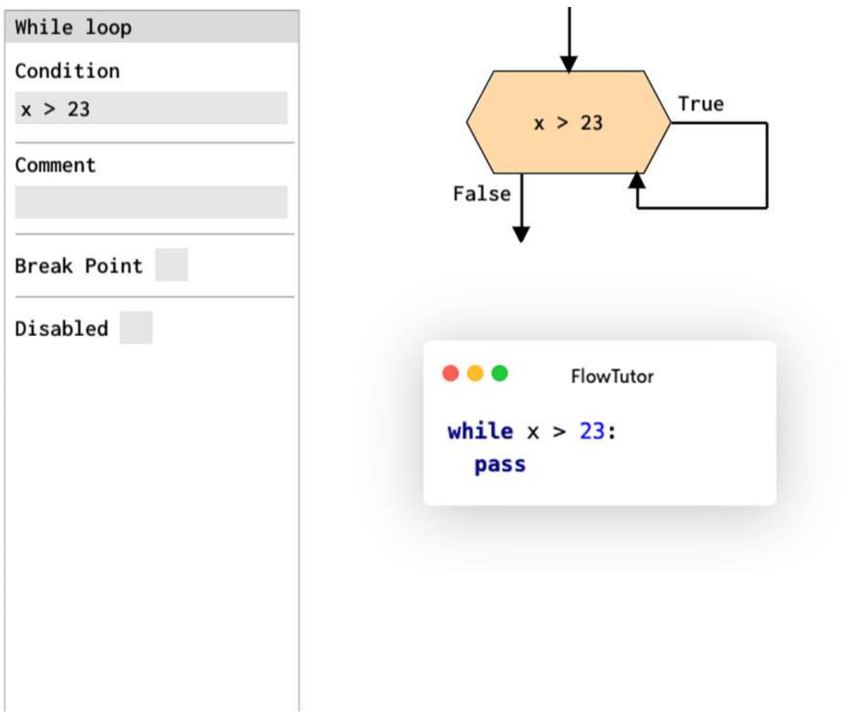
```
some_function()
```

Figure 3.31: Python "Call" node and resulting source code.



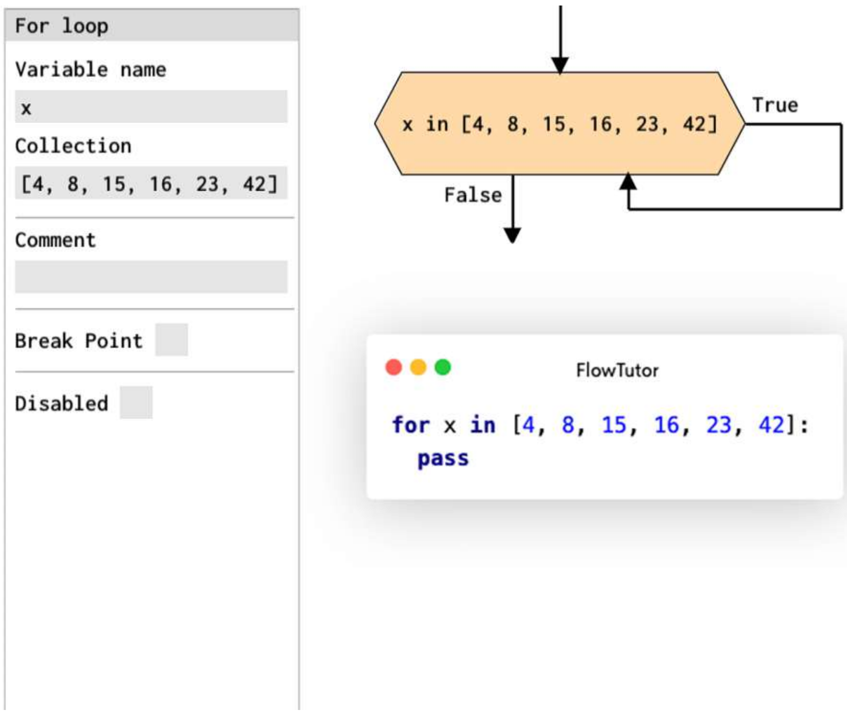
```
if x > y:
  pass
```

Figure 3.32: Python “Conditional” node and resulting source code.



```
while x > 23:
  pass
```

Figure 3.33: Python “While”-loop node and resulting source code.


For loop

Variable name

x

Collection

[4, 8, 15, 16, 23, 42]

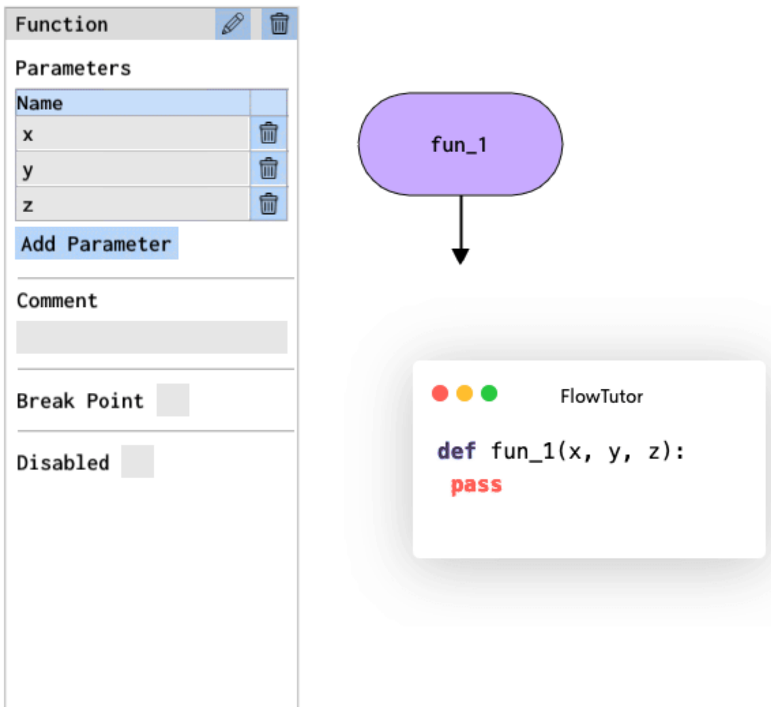
Comment

Break Point

Disabled

```python
for x in [4, 8, 15, 16, 23, 42]:
  pass
```

Figure 3.34: Python "For"-loop node and resulting source code.


Function

Parameters

Name
x
y
z

Add Parameter

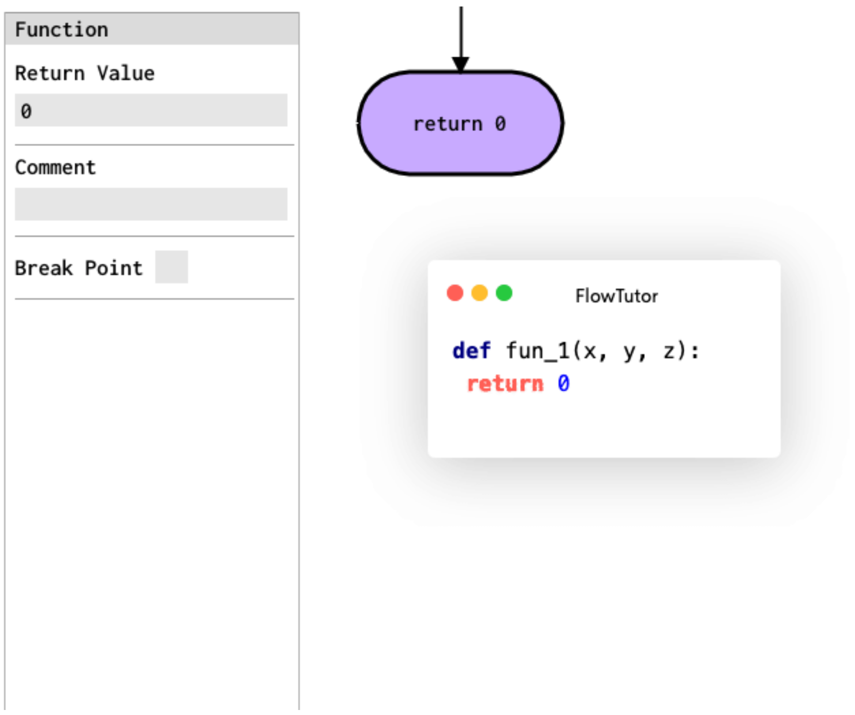
Comment

Break Point

Disabled

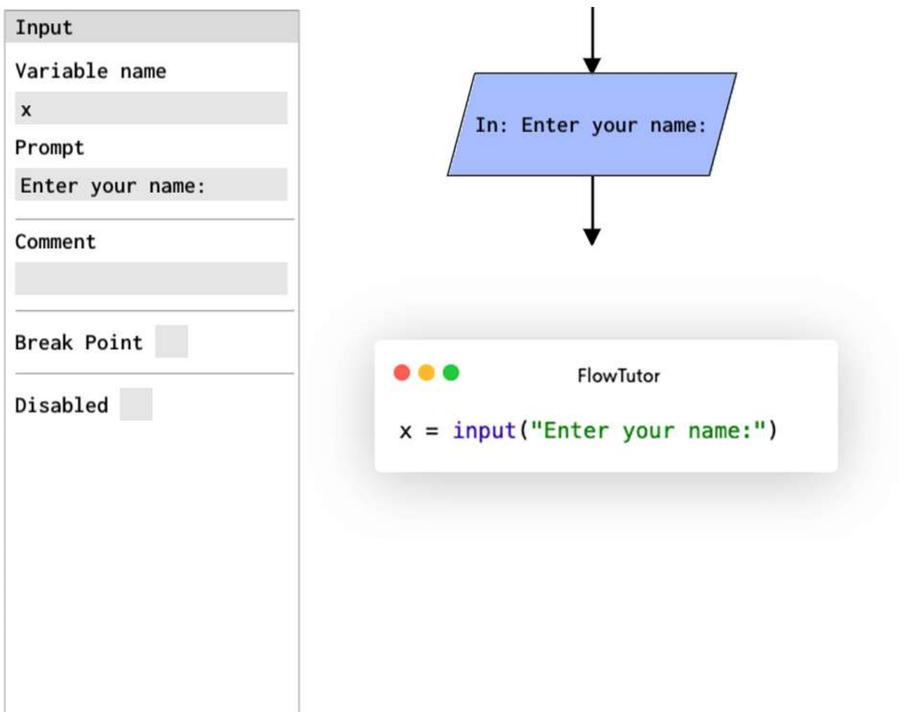
```python
def fun_1(x, y, z):
  pass
```

Figure 3.35: Python "Function Start" node and resulting source code.



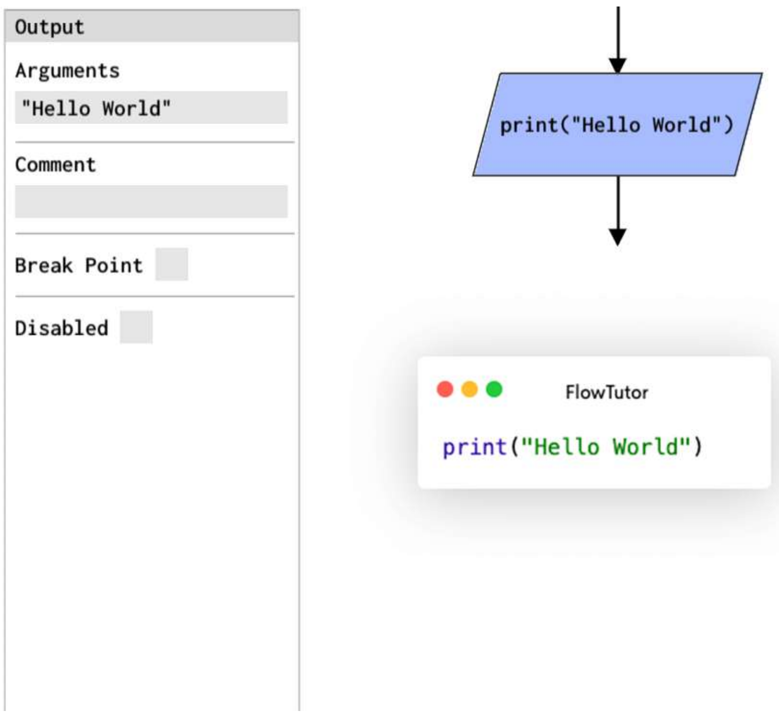
```
def fun_1(x, y, z):
 return 0
```

Figure 3.36: Python "Function End" node and resulting source code.



```
x = input("Enter your name:")
```

Figure 3.37: Python "Input" node and resulting source code.


Output

Arguments

"Hello World"

Comment

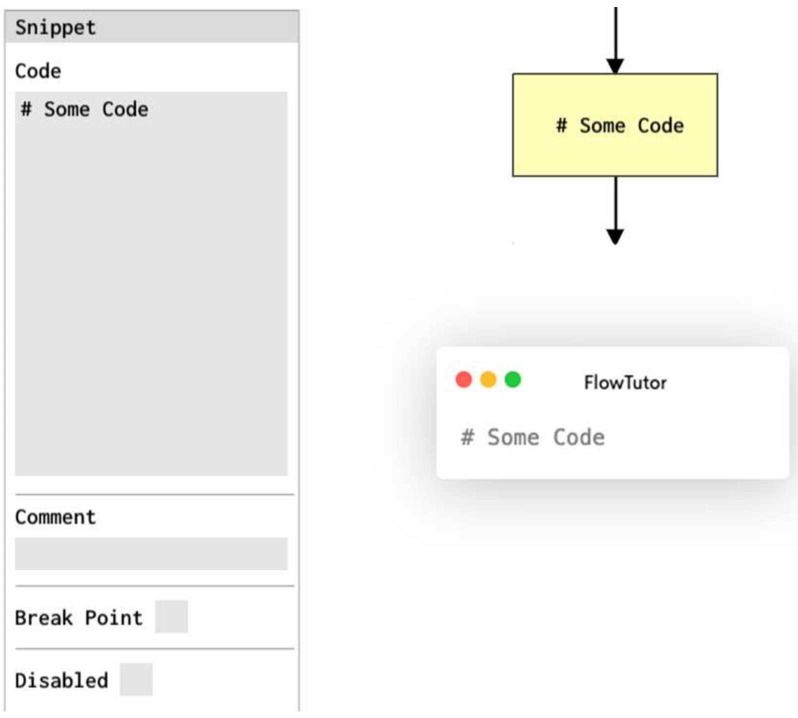
Break Point

Disabled

FlowTutor

```
print("Hello World")
```

Figure 3.38: Python “Output” node and resulting source code.


Snippet

Code

```
# Some Code
```

Comment

Break Point

Disabled

FlowTutor

```
# Some Code
```

Figure 3.39: Python “Snippet” node and resulting source code.

3.3.5 Code Generator

The code generator takes a list of `Flowchart` objects, generates the corresponding source code, and saves it to a file.

This generated code is structured in the following order:

- Module imports
- Type definitions (optional)
- Struct definitions (optional)
- Preprocessor definitions (optional)
- Custom code
- Function declarations (optional)
- Function definitions.
  Depending on whether the selected language has function declarations, the order of the functions is unchanged or the `main` function is inserted at the end, so all other functions are defined before their use.
  The generator iterates over the nodes in the flowcharts, starting with the root node. It recursively resolves the decision and loop nodes to generate the function definitions.

The resulting code file is placed in the working directory.

To facilitate debugging of the application, the code generator assigns the corresponding source-code line numbers back to their originating nodes.

3.3.6 Debugger

Since there is no standardized interface for debugging programs in arbitrary programming languages, we have implemented integrations with the GNU debugger (GDB) for C and the generic Python debugger base class `Bdb`.

To consolidate the difference in controlling different debuggers, we have defined an abstract base class `DebugSession` with concrete implementations for each case.

C Debugger

To debug our programs with GDB, we first need an executable of the program to debug, which has been compiled with debugging symbols turned on. Therefore, before a C-program can be debugged in *FlowTutor*, the user must compile it by clicking on the appropriate button. This starts the compilation process with GCC and the executable is placed in the working directory. Every time a change is made to the program, it must be recompiled.

When the program is executed, the application starts GDB in a sub-process. Normally GDB is used in a command line interface (CLI) environment and to this end, it produces output that is suitable to be read by humans. For the purpose of interpreting its output in a program this presents a challenge, because the output of the debugged program is mixed in with the debugger output, with no clear indication which is which. Luckily, GDB offers a solution for use cases such as this: There exists a mode, which generates machine-readable output, called the GDB Machine Interface (GDBMI). With the GDBMI enabled, program output is provided with a special prefix which allows the program to distinguish between the two.

Our debug session communicates with the GDB sub-process through the GDBMI over a TTY, if available. If there are no TTYs available on the system, as in Windows for example, the input and output of the program itself take place through an external console window.

Breakpoint definitions are written into a separate file which gets loaded into GDB on every change via the `source` command.

Variable assignments are queried on every breakpoint hit or if there is a single step made in the execution. Value types are directly displayed to the user and pointers are resolved to the value to which they point, as this is more pertinent information for the purpose of debugging simple programs, such as our intended use case.

The debug session reports changes to its state, such as execution steps, hit breakpoints, and variable assignments back to the GUI through signals.

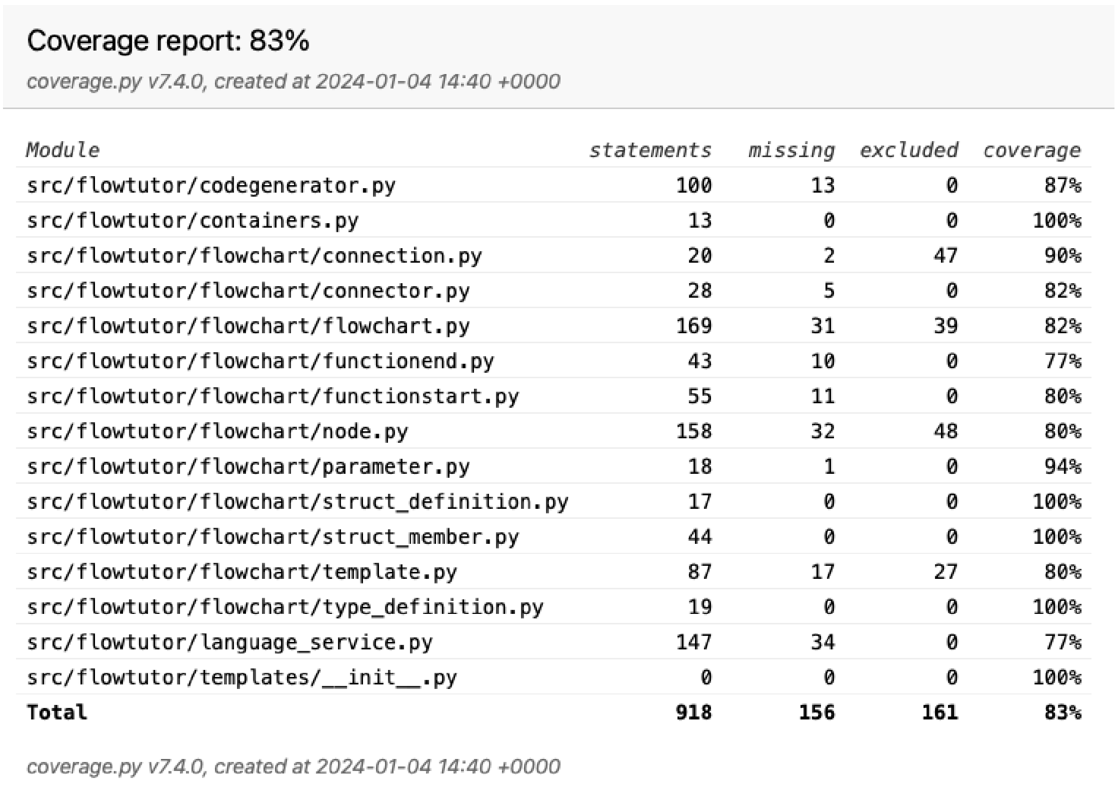
Python Debugger

The *Python Standard Library* provides a debugger framework in the form of the `bdb` module. `Bdb` "[...] handles basic debugger functions, like setting breakpoints or managing execution via the debugger." [Pytb]. Our implementation derives from the `Bdb` base class and handles the interaction with our GUI. The code generated from the flowchart is run in the same process as the application through the `run` method of the debugger class derived from `Bdb`.

We do not want to block our application with the execution of the program we are debugging and therefore a separate thread is started in which the application runs separately from the UI.

Since the users usually do not have access to the console input and output facilities, from which the application is started, we redirect `stdout`, `stdin`, and `stderr` to a Python `multiprocessing.Queue` object which we wrapped inside a separate class. This allows us to read from and write to the queue from the GUI thread.

When a breakpoint is hit, a `threading.Barrier` object is instantiated. This `Barrier` lets the debugger thread wait until the `interact` method is called from the GUI thread. Every time a line is hit, either from a breakpoint or by stepping through the code, the debugger reads the local variables from the current stack frame and sends them to the GUI to be displayed to the user. For the main stack frame of the program, there are some global variables which would pollute our application with unnecessary information, so we filter those out.



Coverage report: 83%

coverage.py v7.4.0, created at 2024-01-04 14:40 +0000

Module	statements	missing	excluded	coverage
src/flowtutor/codegenerator.py	100	13	0	87%
src/flowtutor/containers.py	13	0	0	100%
src/flowtutor/flowchart/connection.py	20	2	47	90%
src/flowtutor/flowchart/connector.py	28	5	0	82%
src/flowtutor/flowchart/flowchart.py	169	31	39	82%
src/flowtutor/flowchart/functionend.py	43	10	0	77%
src/flowtutor/flowchart/functionstart.py	55	11	0	80%
src/flowtutor/flowchart/node.py	158	32	48	80%
src/flowtutor/flowchart/parameter.py	18	1	0	94%
src/flowtutor/flowchart/struct_definition.py	17	0	0	100%
src/flowtutor/flowchart/struct_member.py	44	0	0	100%
src/flowtutor/flowchart/template.py	87	17	27	80%
src/flowtutor/flowchart/type_definition.py	19	0	0	100%
src/flowtutor/language_service.py	147	34	0	77%
src/flowtutor/templates/__init__.py	0	0	0	100%
Total	**918**	**156**	**161**	**83%**

coverage.py v7.4.0, created at 2024-01-04 14:40 +0000

Figure 3.40: Testy-coverage report.

3.4 Testing

The first intention was to base the entire development process on the methodology of test-driven-development (TDD) [Bec02]. With this approach, the automated test cases are written before the actual implementation. The feature is considered complete when all the predefined tests complete successfully.

The heavy focus on the GUI and graphical elements of *FlowTutor*, made TDD infeasible to be applied to the whole process. The GUI framework used does not have provisions for automated tests and tests based on image comparison of screenshots were considered not to have enough advantages to overcome their immense maintenance requirements.

GUI testing was instead carried out through exploratory testing and the creation of checklists of test cases.

TDD was not abandoned fully and was employed wherever automated tests could be used efficiently. This was the case for the logic of the flowchart nodes, such as how they are added and removed from the flowchart, as this was separated from the graphical representation. The source code generator also contains extensive test cases.

With insignificant code like system utility methods and GUI setup code excluded, we have a test coverage of 83% in the latest build (see Figure 3.40).

3.5 Continuous Delivery

Continuous Integration (CI) was first written about by Beck in 1999 [Bec99] and is the practice of frequently integrating code changes into a shared repository. With every commit to the source code repository, automated tests run on the CI server, in order to detect regressions early on.

Continuous Delivery (CD) extends CI by automating the entire delivery pipeline, including testing, deployment, and releases [HF10]. The goal of continuous delivery is to ensure that software can be released to production at any time with minimal manual intervention, reducing the time and risk associated with the release process.

An overview of our CD process can be seen in Figure 3.41, with manual steps depicted in orange and automated steps carried out by the CI server in blue.

New features and bug fixes first get developed in their own separate feature branch. These branches are supposed to be small in scope, so they can be merged frequently back into the development branch. To avoid a broken CI pipeline, testing should be carried out locally first.

Each push to the repository triggers the CI pipeline, including automated tests and style checks. For successful runs, a test-coverage report is generated. In order to avoid an excessive number of releases, the release process is triggered by the creation of a release version tag in the git repository. This should only be done if the CI pipeline runs through successfully and all manual tests are successful.

The server builds the Python package with the appropriate version number and deploys it automatically to PyPI.

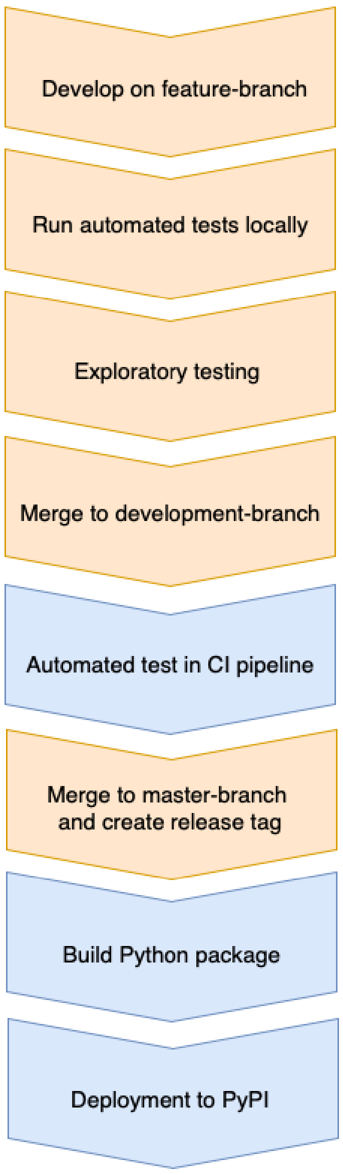

Develop on feature-branch
Run automated tests locally
Exploratory testing
Merge to development-branch
Automated test in CI pipeline
Merge to master-branch and create release tag
Build Python package
Deployment to PyPI

Figure 3.41: Continuous delivery process, with manual steps depicted in orange and automated steps carried out by the CI server in blue.

CHAPTER 4

Evaluation

During the development of *FlowTutor*, an initial round of informal evaluations was performed with tutors of the introductory programming lecture at the *Institute of Microelectronics*. Four of the tutors used *FlowTutor* to solve a simple programming task and performed some informal exploratory testing. The feedback was collected in the form of qualitative statements and bug reports and was incorporated into subsequent iterations of the application.

Later in the evaluation process, there was a wider evaluation, in the form of a voluntary exercise, where students provided quantitative as well as qualitative feedback. This evaluation is described in the following sections.

4.1 Methods

4.1.1 Participants

The participants are students of the bachelor program of electrical engineering, who attended an introductory programming lecture before taking part in the evaluation. There were eleven students in total who completed the assignment and responded to our questionnaire and feedback questions.

4.1.2 Design

The study was a quasi-experimental, empirical mixed-methods study. The participants were separated into five groups and solved a small programming task using *FlowTutor*.

The scale used for measuring usability is the *System Usability Scale (SUS)*. Described by its inventor as “a simple, ten-item scale giving a global view of subjective assessments of usability.” [Bro95].

The ten statements were rated by the participants with a value of 1 (strongly disagree) to 5 (strongly agree).

Based on the supplied values a value between 0 and 100 is calculated, which is meant to objectively evaluate the usability of the application. The exact questions can be found in Appendix A.

The workload was measured with the NASA Task Load Index (TLX) [HS88], which is described by its inventors as “the results of a multi-year research program to identify the factors associated with variations in subjective workload.” [HS88].

Similarly to the SUS, the participants answer six questions with a value between 1 and 5. Based on the answers a value between 0 and 100 is calculated, which indicates the workload with the application. The exact questions can be found in Appendix B.

Besides these two indices, we added questions for students to self-evaluate their programming abilities and we posed the following questions and collected the responses as statements:

1. Do you feel *FlowTutor* made solving the assignment easier, or was it a hindrance? Please elaborate on your answer.
2. Compare your experience using *FlowTutor* with your experience programming with a text editor or IDE.
3. Was there some missing functionality that prevented you from accomplishing a desired result, or forced you to use a workaround?
4. Which feature or set of features needs the most improvement to provide a better experience for students?

4.2 Results

Detailed results of the questionnaires and feedback questions are provided in Appendix C.

4.2.1 System Usability Scale (SUS)

The total average SUS score over all participants in our evaluation was 55.45 on a scale from 0 to 100, where 0 is least usable and 100 is most usable. In order to be able to draw conclusions from this value, it must be classified. Although it is clear that a value cannot be below 0 and not above 100, it is not clear how to classify other values in between. What constitutes acceptable user-friendliness and in which range is the application to be classified as unusable?

Bangor et al. attempted to find an answer these questions [BKM09]. To this end, an eleventh question was added to almost 1000 SUS surveys:

"Overall, I would rate the user-friendliness of this product as: 1 Worst Imaginable, 2 Awful, 3 Poor, 4 OK, 5 Good, 6 Excellent, 7 Best Imaginable"

The results showed that the answers to the added question correlate well with the SUS scores ($r = 0.822$).

Based on these results, the values obtained can be interpreted more accurately. As can be seen in Figure 4.1, our participants ranked the usability of *FlowTutor* in the marginally acceptable range. An explanation for this low score, which stands to reason becomes apparent when looking at the results for the questions about programming experience. All of the participants report being at least somewhat experienced with programming, making it inefficient to have to navigate flowchart creation in *FlowTutor*, when they are already able to write the source code themselves.

While the sample is too small to draw accurate statistical conclusions, we suspect that the SUS score might inversely correlate with prior programming experience. When divided into two groups, with one having little to no prior Python experience (Answers 1 and 2) and the other having some or a lot of experience (Answer 3 and above), an independent-samples t test suggests that there is a significant difference in the SUS scores between the groups ($t(9) = 2.4209$, $p < 0.05$). Although it supports our hypothesis, the small sample size of $N = 11$ should be kept in mind with this result.

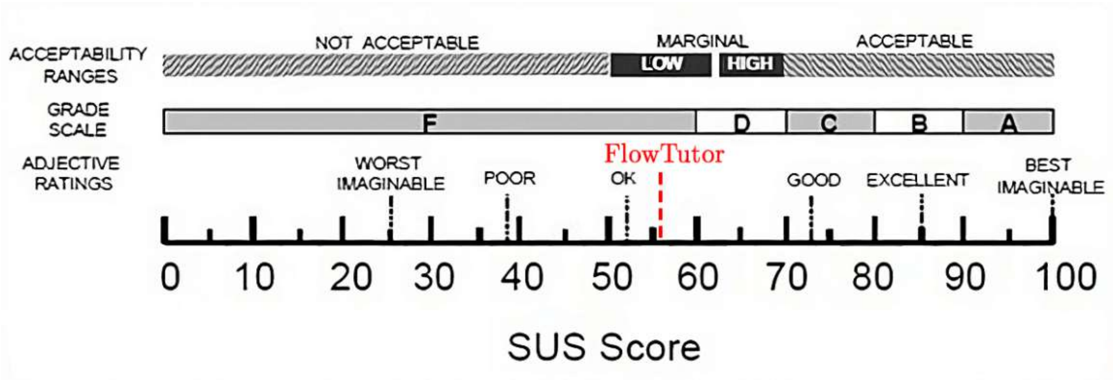


Figure 4.1: FlowTutor SUS score (marked in red) compared to the grade rankings of SUS scores, originally published in [BKM09].

4.2.2 NASA Task Load Index (TLX)

The total average TLX in our evaluation was 28.79 on a scale of 0 to 100 with 100 being the highest workload and therefore the least desirable.

It could be argued that this result has a downward bias, since physical and temporal demands are not relevant in our application, because there is only negligible physical effort involved in any desktop computer application and the participants did not have limiting time restrictions in their assignment, which could be solved over the course of several months. When we exclude these factors from the analysis, the TLX comes to 31.25.

There are similar questions regarding classification for the TLX as for the SUS. How can we classify this numerical value? Hertzum attempted to aid researchers with this problem by conducting a meta-analytic review of 556 studies using the TLX to gather task load data [Her21]. His paper supplies reference values to use as a benchmark for our results.

As can be seen in Figure 4.2, FlowTutor has a relatively low workload value across all the subscales of the TLX. As previously mentioned, the moderately high value in the inefficiency scale might be explained by the prior programming experience of the participants, who are not the target audience of this tool

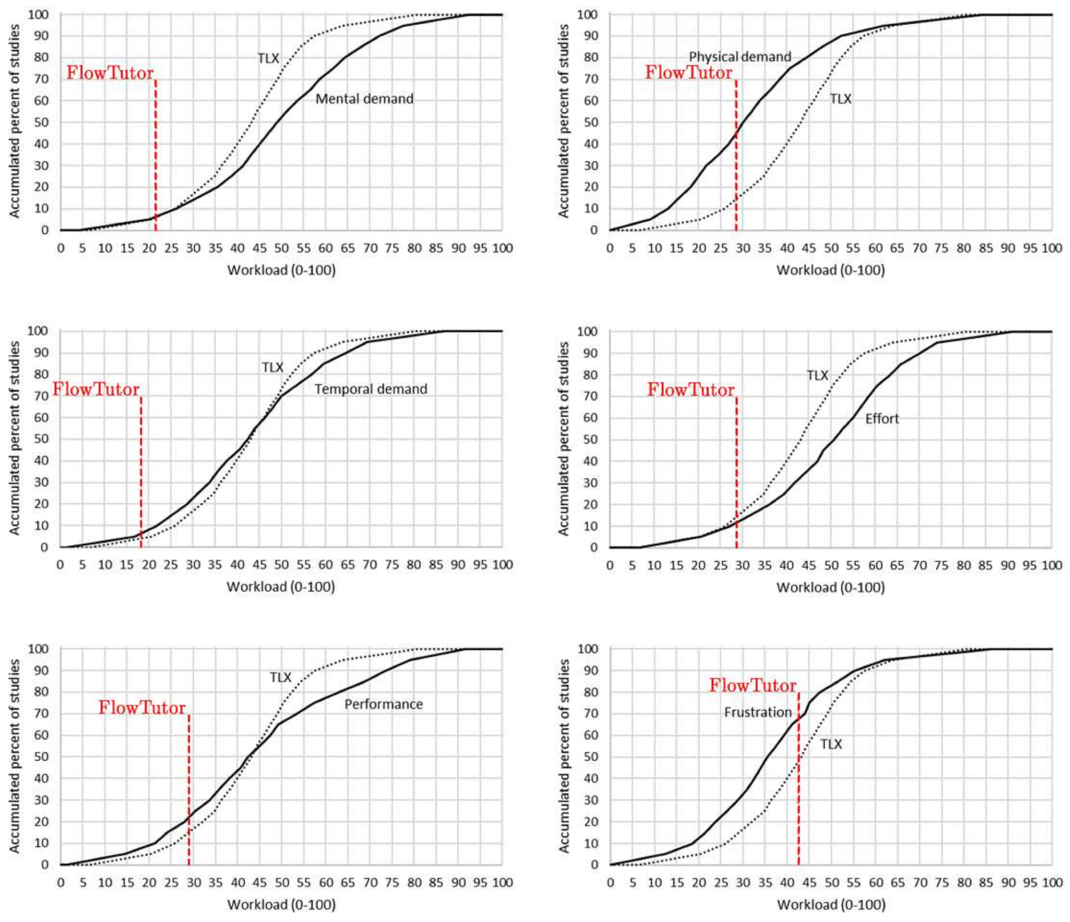


Figure 4.2: "Distribution of the six TLX subscales (solid lines) and the TLX score (dotted line), N = 556 studies", originally published in [Her21]. Results from our evaluation additionally marked in red.

The answers to the free-form questions were received mostly as prose and we summarized the answers into their essential meaning and grouped them into positive statements, negative statements, and improvement suggestions, which are provided in Appendix C. As most participants had experience programming, a common theme in their answers was, that they felt held back by having to draw the flowcharts instead of writing source code of their own, but they could see the value of visualizing difficult-to-understand algorithms.

CHAPTER 5

Future Work

While we have confirmed the benefits of FlowTutor for beginners to programming, there are nevertheless several areas of *FlowTutor* for which there is room for improvement.

Users are accustomed to certain basic features, which are regarded as self-evident basic functionality across applications, such as the ability to copy and paste elements and being able to undo their actions at any point. The mentioned features are missing from the current version of *FlowTutor*, which are fundamental features of software tools as was mentioned by the participants in our evaluation. Implementing them and focusing more on quality-of-life features, in general, would make the tool more attractive for students.

Getting started with *FlowTutor* could be made easier, with improved documentation and by providing tutorials, either in video form or with text and pictures.

Our predefined templates focused on the basics, but further node types could be defined to help focus students on various aspects of programming assignments. An example was reflected in one feedback we obtained during the evaluation, where a student wished there were nodes for string manipulation operations because he felt like he needed to know Python syntax to be able to solve the assignment.

More varying language definitions to enable other programming languages could also be created. To be able to debug other languages, a generic solution to integrate external debuggers would be a significant improvement.

Currently *FlowTutor* only supports a procedural paradigm for the most part, the templating system and GUI could be improved to support other programming constructs more intuitively, e.g., classes.

Visualization of the program state during debugging is only implemented in a rudimentary way. Better solutions, especially graphical representations of references, like arrays and pointers, could be tremendously helpful for demonstrating and understanding complex algorithms.

Participants in the evaluation expressed the desire for an application that would reverse the principle of *FlowTutor*: The flowchart should be generated based on source code written by the user.

The evaluation served a useful purpose in gathering valuable feedback for future developments as well as establishing a quantitative baseline for future evaluations. More long-term studies should be carried out, comparing the efficacy and usability of our application, with an approach which relies solely on textual programming.

APPENDIX A

System Usability Scale (SUS)

	Strongly Disagree				Strongly Agree
I think that I would like to use FlowTutor frequently.	☐	☐	☐	☐	☐
I found FlowTutor unnecessarily complex.	☐	☐	☐	☐	☐
I thought FlowTutor was easy to use.	☐	☐	☐	☐	☐
I think that I would need the support of a technical person to be able to use FlowTutor.	☐	☐	☐	☐	☐
I found the various functions in FlowTutor were well integrated.	☐	☐	☐	☐	☐
I thought there was too much inconsistency in FlowTutor.	☐	☐	☐	☐	☐
I would imagine that most people would learn to use FlowTutor very quickly.	☐	☐	☐	☐	☐
I found FlowTutor very cumbersome to use.	☐	☐	☐	☐	☐
I felt very confident using FlowTutor.	☐	☐	☐	☐	☐
I needed to learn a lot of things before I could get going with FlowTutor.	☐	☐	☐	☐	☐

APPENDIX B

NASA Task Load Index (TLX)

How much mental and perceptual activity was required? Was the assignment easy or demanding, simple or complex, exacting or forgiving?				
Very low				Very high
☐	☐	☐	☐	☐

How much physical activity was required (e.g. pushing, pulling, turning, controlling, activating. etc.)? Was the assignment easy or demanding, slow or brisk, slack or strenuous, restful or laborious?				
Very low				Very high
☐	☐	☐	☐	☐

How much time pressure did you feel due to the rate or pace at which the assignment or assignment elements occurred? Was the pace slow and leisurely or rapid and frantic?				
Very low				Very high
☐	☐	☐	☐	☐

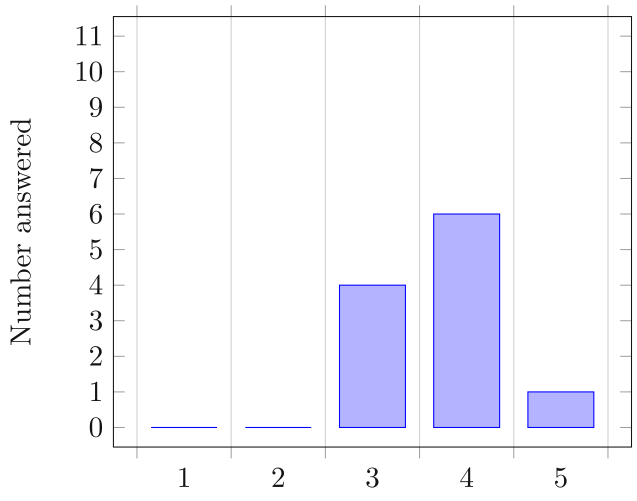
How successful do you think you were in accomplishing the goals of the assignment set by the instructors (or yourself)? How satisfied were you with your performance in accomplishing these goals?				
Perfect				Failure
☐	☐	☐	☐	☐

How hard did you have to work (mentally and physically) to accomplish your level of performance?				
Very low				Very high
☐	☐	☐	☐	☐

How insecure, discouraged, irritated, stressed, and annoyed versus secure, gratified, content, relaxed and complacent did you feel during the assignment?				
Very low				Very high
☐	☐	☐	☐	☐

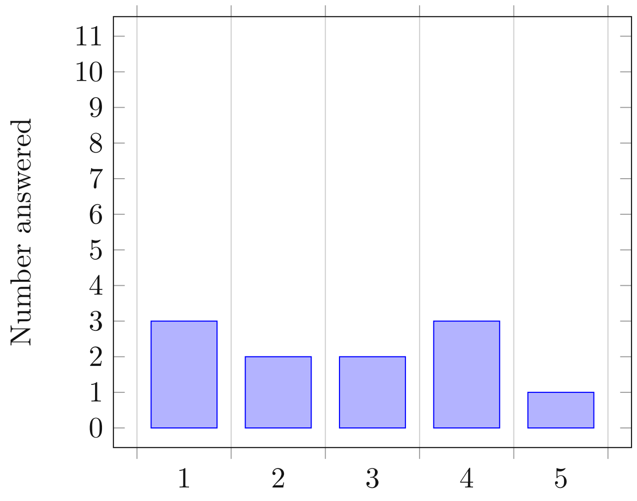
APPENDIX C

Evaluation Results



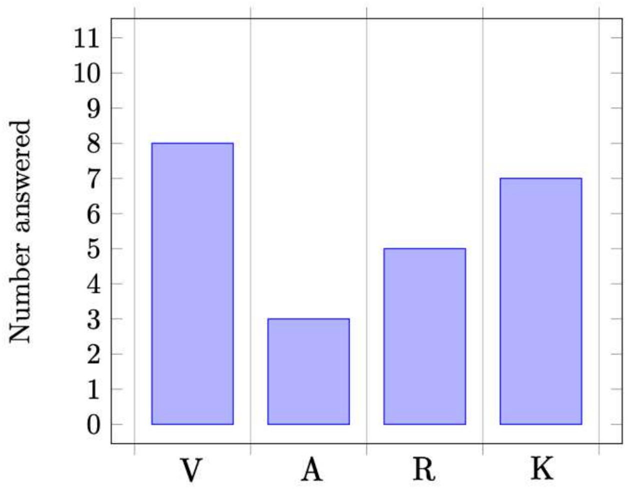
“How would you rate your experience with programming in general before attending the lecture?”

(1) No experience – (5) Very experienced



“How would you rate your experience with programming in Python before attending the lecture?”

(1) No experience – (5) Very experienced

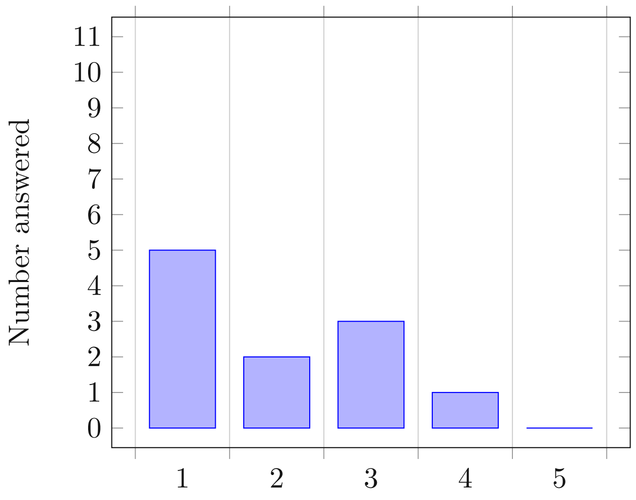


“Based on VARK model of learning styles, which style or styles do you closest identify with? (Multiple choices allowed)”

(V) Visual, (A) Aural, (R) Read/Write, (K) Kinesthetic

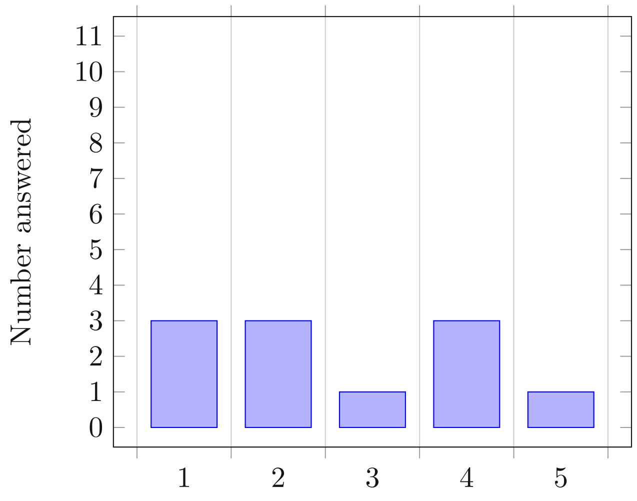
C.1 System Usability Scale (SUS)

Total average SUS score: **55.45**



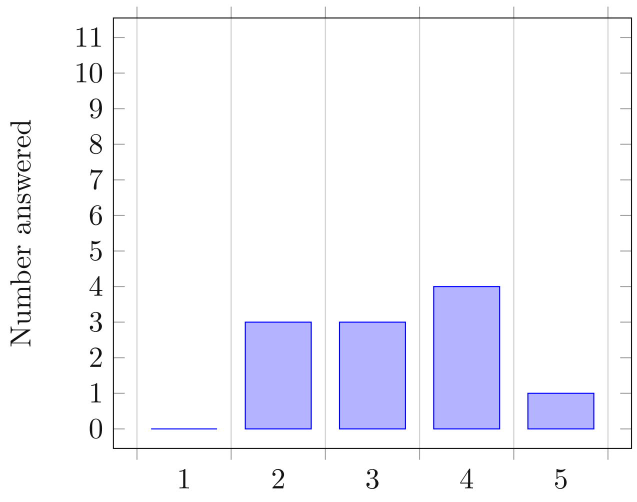
“I think that I would like to use FlowTutor frequently.”

(1) Strongly disagree – (5) Strongly agree



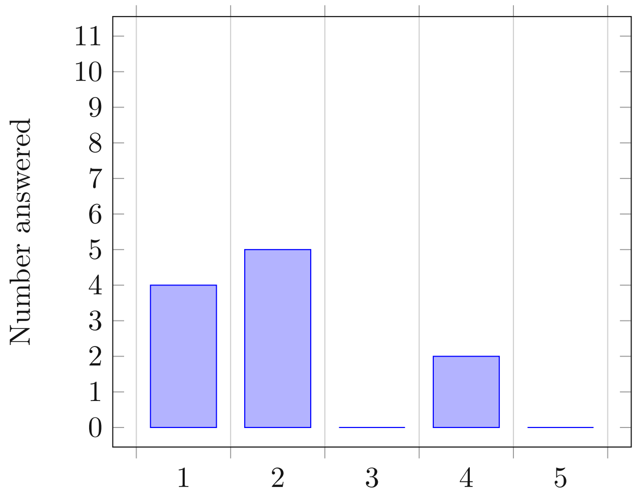
"I found FlowTutor unnecessarily complex."

(1) Strongly disagree – (5) Strongly agree



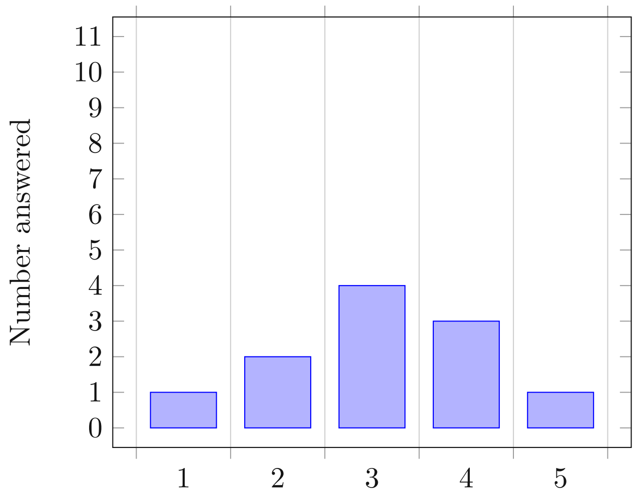
"I thought FlowTutor was easy to use."

(1) Strongly disagree – (5) Strongly agree



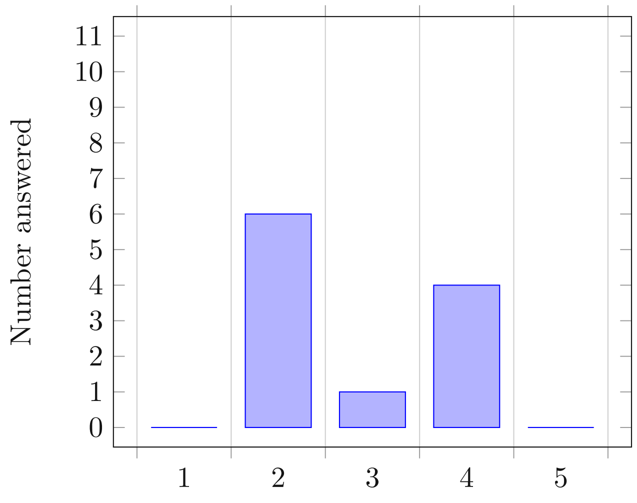
"I think that I would need the support of a technical person to be able to use FlowTutor."

(1) Strongly disagree – (5) Strongly agree



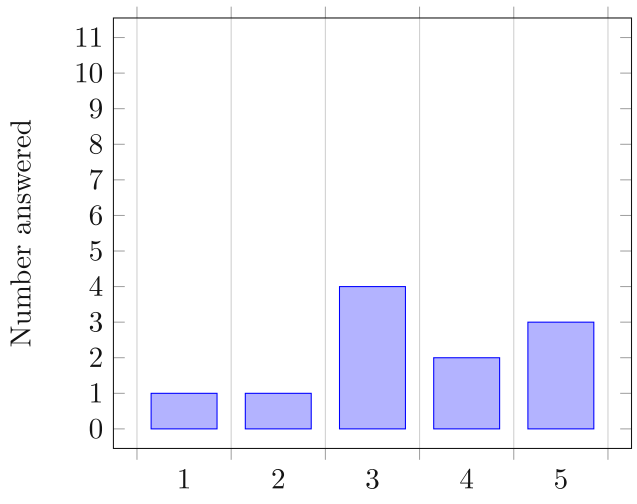
"I found the various functions in FlowTutor were well integrated."

(1) Strongly disagree – (5) Strongly agree



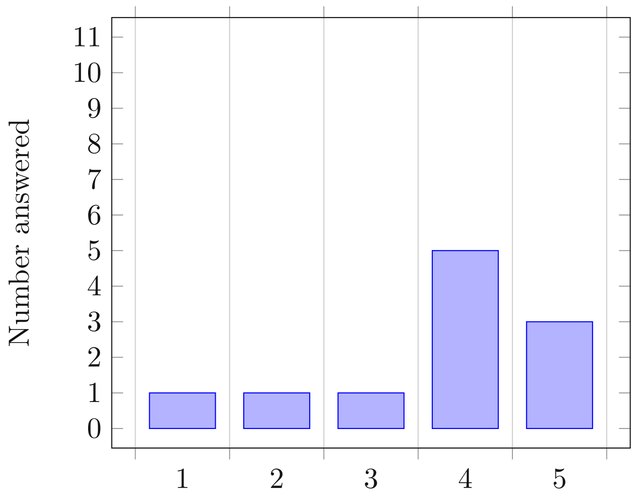
“I thought there was too much inconsistency in FlowTutor.”

(1) Strongly disagree – (5) Strongly agree



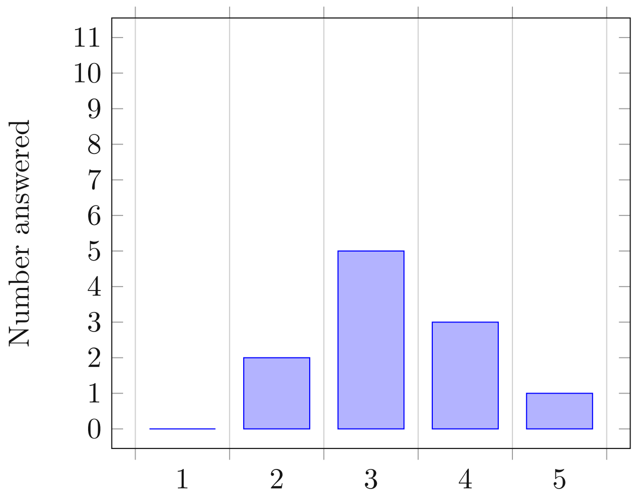
“I would imagine that most people would learn to use FlowTutor very quickly.”

(1) Strongly disagree – (5) Strongly agree



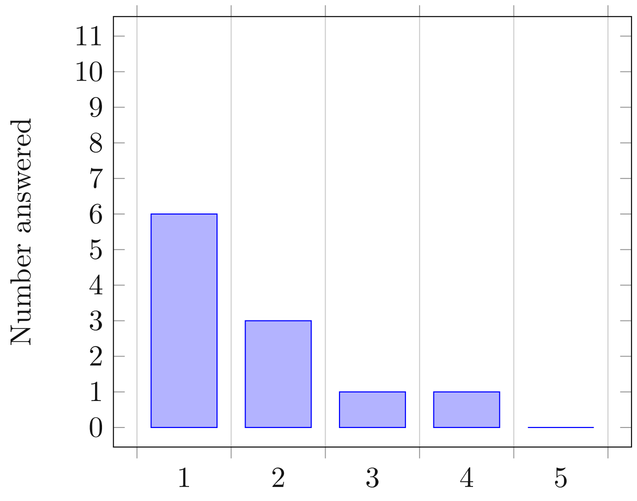
"I found FlowTutor very cumbersome to use."

(1) Strongly disagree – (5) Strongly agree



"I felt very confident using FlowTutor."

(1) Strongly disagree – (5) Strongly agree



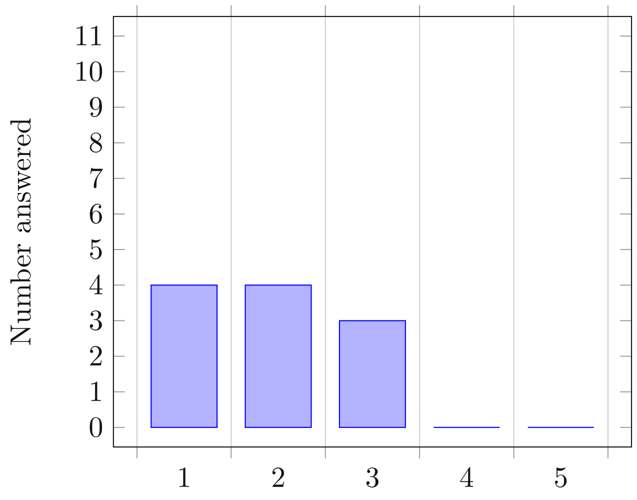
"I needed to learn a lot of things before I could get going with FlowTutor."

(1) Strongly disagree – (5) Strongly agree

C.2 NASA Task Load Index (TLX)

Total average TLX: **28.79**

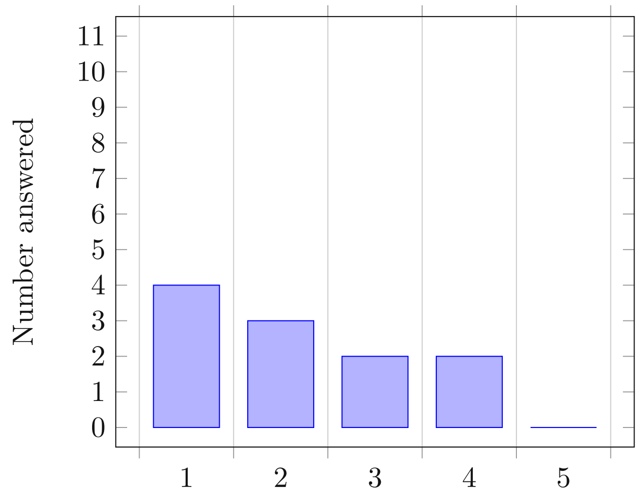
Total average TLX (excluding physical and temporal demand): **31.25**



"How much mental and perceptual activity was required? Was the assignment easy or demanding, simple or complex, exacting or forgiving?"

(1) Very low – (5) Very high

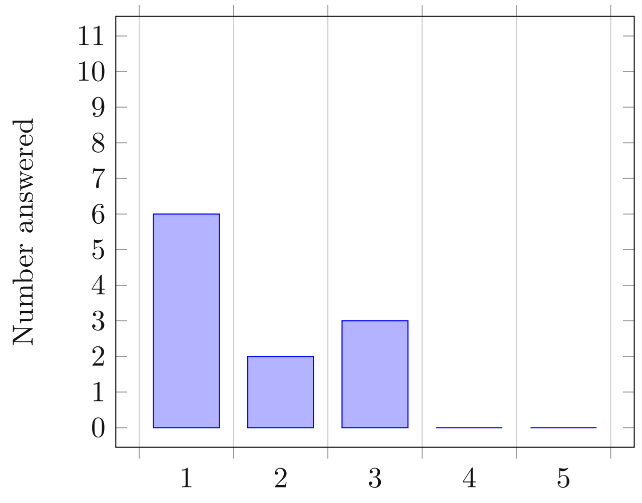
Total average mental demand: **22.73**



"How much physical activity was required (e.g. pushing, pulling, turning, controlling, activating. etc.)? Was the assignment easy or demanding, slow or brisk, slack or strenuous, restful or laborious?"

(1) Very low – (5) Very high

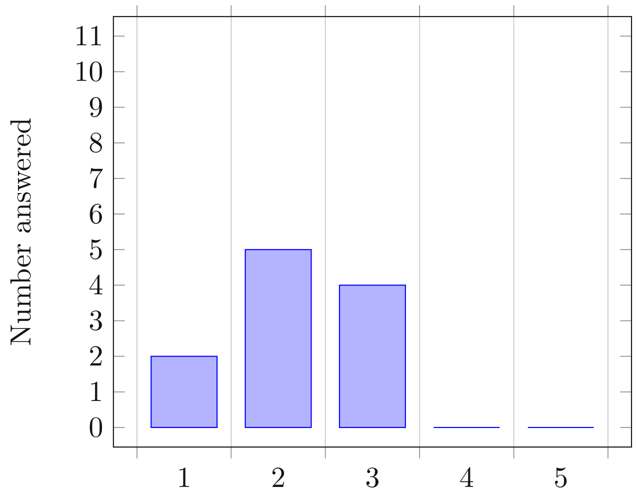
Total average physical demand: **29.55**



"How much time pressure did you feel due to the rate or pace at which the assignment or assignment elements occurred? Was the pace slow and leisurely or rapid and frantic?"

(1) Very low – (5) Very high

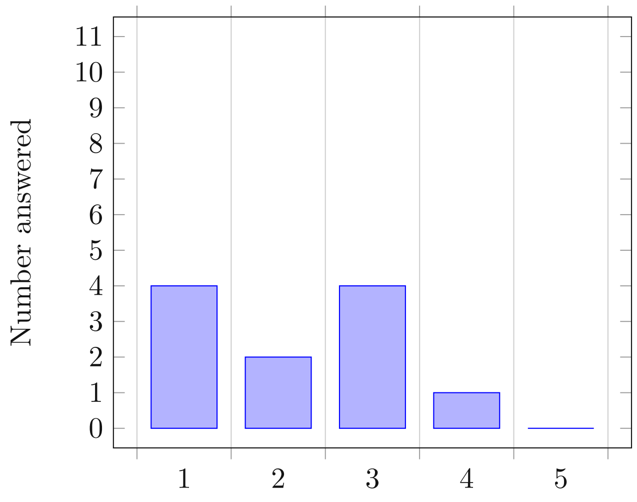
Total average temporal demand: **18.18**



“How successful do you think you were in accomplishing the goals of the assignment set by the instructors (or yourself)? How satisfied were you with your performance in accomplishing these goals?”

(1) Perfect – (5) Failure

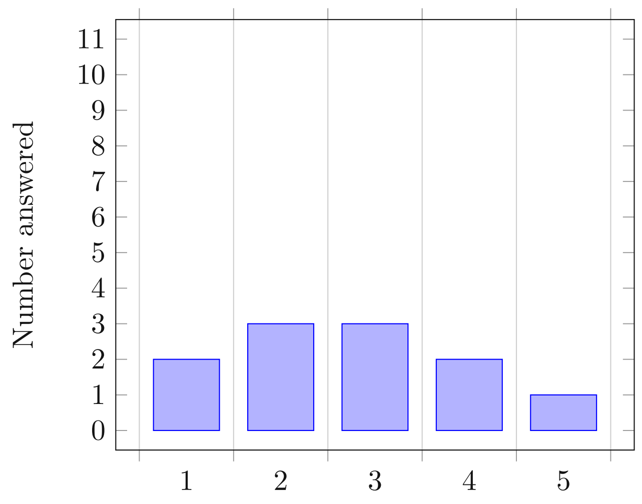
Total average performance (lower is better): **29.55**



“How hard did you have to work (mentally and physically) to accomplish your level of performance?”

(1) Very low – (5) Very high

Total average effort: **29.55**



"How insecure, discouraged, irritated, stressed, and annoyed versus secure, gratified, content, relaxed and complacent did you feel during the assignment?"

(1) Very low – (5) Very high

Total average frustration: **43.18**

C.3 Feedback Questions

Positive statements in order of their prevalence in our evaluation:

Statement	Occurrences
There are fewer opportunities for syntax errors and typos.	3
The flowchart gives a nice overview of the algorithm the program represents.	3
The flowcharts help to visualize the program flow.	3
It is easier to get started than with a textual IDE, due to its focused feature set.	1
Debugging is easier to understand than with a textual IDE.	1

Negative statements in order of their prevalence in our evaluation:

Statement	Occurrences
Developing feels slower than with a text editor.	8
It is frustrating to have to manipulate the flowchart, when the necessary change in the source code would be obvious.	5
There are difficulties with the discoverability of functionality.	3
Too much knowledge of the language syntax is necessary to accomplish a task.	2
It would be hard to keep an overview of larger projects in the flowchart representation.	2
There are graphical glitches in the UI and flowchart.	2
The code execution is slower than if the program would be executed outside of FlowTutor.	2
There is too little documentation.	1
There is no existing community where one could turn to for questions.	1
The use of the mouse is necessary.	1
One needs to adjust the nodes constantly to keep the flowchart visually tidy.	1
Comments are not visible in the flowchart.	1
The separate tabs for functions is cumbersome because of the need to constantly switch between them.	1
Error messages are less helpful than in console.	1

Improvement suggestions in order of their prevalence in our evaluation:

Suggestion	**Occurrences**
Add a delete button for nodes in the UI.	4
Add the ability to copy and paste nodes and groups of nodes.	3
Add the ability to zoom the flowchart view.	3
Provide more UI controls for defining complex parameters like conditional statements.	3
Add undo and redo functionality.	2
Reverse the principle of the application: Let the user write the source code and generate the flowchart.	2
Provide a tutorial on how to get started with FlowTutor.	2
Add a grid to arrange and align nodes more easily.	2
Show code comments in the graphical view of the flowchart.	2
Add affordances related to the file save status.	1
Add automatic saving.	1
Add ability to insert nodes with drag and drop.	1
Manipulate node parameters directly in the flowchart view, instead of the sidebar.	1
Provide the ability to collapse groups of nodes.	1
Highlight the corresponding source code, when highlighting the node in the flowchart view.	1
Add tooltips throughout the application.	1
Provide a way to run the program in a slowed down manner, to be able to follow the execution visually.	1
Add an area to put nodes for later use, like a tray.	1
Add dedicated node types for loop commands, i.e., `break` and `continue`.	1
Add syntax highlighting in the source code view.	1

List of Figures

Bibliography

[ans70] ANSI X3.5 - Flowchart Symbols and their Usage in Information Processing. Standard, American National Standards Institute, 1970.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[Bec02] Kent Beck. *Test Driven Development. By Example*. Addison-Wesley, 2002.

[BK00] Lynne P. Baldwin and Jasna Kuljis. Visualisation techniques for learning and teaching programming. In *Proceedings of the 22nd International Conference on Information Technology Interfaces*, pages 83–90, 2000.

[BKM09] Aaron Bangor, Philip Kortum, and James Miller. Determining what individual sus scores mean: adding an adjective rating scale. 4(3):114–123, 2009.

[BM95] Margaret M. Burnett and David W. McIntyre. Visual programming. *Computer*, 28:14–14, 1995.

[BM99] Jean Bézivin and Pierre-Alain Muller. Uml: The birth and rise of a standard modeling notation. In *The Unified Modeling Language.«UML»'98: Beyond the Notation: First International Workshop, Mulhouse, France*, pages 1–8. Springer, 1999.

[Bro95] John Brooke. Sus: A quick and dirty usability scale. *Usability Evaluation in Industry*, 189, 11 1995.

[Buna] Bundesministerium für Bildung, Wissenschaft und Forschung. Die neue TU für Digitalisierung und digitale Transformation entsteht in Linz. `https://www.bmbwf.gv.at/Themen/HS-Uni/Aktuelles/TU-Linz.html`. Accessed: 2024-02-18.

[Bunb] Bundesministerium für Bildung, Wissenschaft und Forschung. Digitale Grundbildung. `https://www.bmbwf.gv.at/Themen/schule/zrp/dibi/dgb.html`. Accessed: 2024-02-18.

[Cal92] Ben A. Calloni. Baccii: An iconic, syntax-directed windows system for teaching procedural programming. Master's thesis, Texas Tech University, 1992.

[CBH97] Ben A. Calloni, Donald J. Bagert, and H. Paul Haiduk. Iconic programming proves effective for teaching the first year programming sequence. In *Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '97, page 262–266. Association for Computing Machinery, 1997.

[Cha70] Ned Chapin. Flowcharting with the ansi standard: A tutorial. *ACM Computing Surveys*, 2(2):119–146, 1970.

[CM05] Stephen Chen and Stephen Morris. Iconic programming for flowcharts, java, turing, etc. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '05, page 104–107. Association for Computing Machinery, 2005.

[Coh04] Mike Cohn. *User stories applied: For agile software development*. Addison-Wesley, 2004.

[Coo15] Devin D Cook. Flowgorithm: Principles for teaching introductory programming using flowcharts. In *Proceedings of the American Society of Engineering Education Pacific Southwest Conference*, pages 158–167, 2015.

[Cro92] Mary Croarken. The emergence of computing science research and teaching at cambridge, 1936-49. *IEEE Annals of the History of Computing*, 14(4):10–15, 1992.

[CWHH04] Martin C. Carlisle, Terry A. Wilson, Jeffrey W. Humphries, and Steven M. Hadfield. Raptor: Introducing programming to non-majors with flowcharts. *Journal of Computing Sciences in Colleges*, 19(4):52–60, 2004.

[Fac19] Fachverband des Tischlerhandwerks Nordrhein-Westfalen. Digitalisierung im Tischlerhandwerk. `https://www.tischler.nrw/fileadmin/lv_nrw/file/innovation_technologie/2020-Auswertung_Digi-Umfrage.pdf`, 2019. Accessed: 2024-02-18.

[Fow08] Martin Fowler. Inversion of control containers and the dependency injection pattern. `http://www.martinfowler.com/articles/injection.html`, 2008. Accessed: 2024-02-18.

[GA16] Lidia Gorodniaia and Tatiana Andreyeva. Study of programming paradigms. In *INTED2016 Proceedings*, 10th International Technology, Education and Development Conference, pages 7482–7491. IATED, 2016.

[Gaj18] R. Robert Gajewski. Algorithms, programming, flowcharts and flowgorithm. *E-Learning and Smart Learning Environment for the Preparation of New Generation Specialists*, 10:393–408, 2018.

[GG21] Frank Gilbreth and Lillian Gilbreth. Process charts: First steps in finding the one best way to do work. In *Transactions of the American Society of Mechanical Engineers*, pages 1029–1043, 1921.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1 edition, 1994.

[GM15] Daniela Giordano and Francesco Maiorana. Teaching algorithms: Visual language vs flowchart vs textual language. In *2015 IEEE Global Engineering Education Conference (EDUCON)*, pages 499–504, 2015.

[GT84] Ephraim P. Glinert and Steve Tanimoto. Pict: An interactive graphical programming environment. *Computer*, 17(11):7–25, 1984.

[GvN47] Herman Heine Goldstine and John von Neumann. Planning and coding of problems for an electronic computing instrument. Technical report, Institute for Advanced Study Princeton, 1947.

[Her21] Morten Hertzum. Reference values and subscale patterns for the task load index (tlx): a meta-analytic review. *Ergonomics*, 64(7):869–878, 2021.

[HF10] Jez Humble and David G. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.

[HGB$^+$13] Brian Harvey, Daniel D. Garcia, Tiffany Barnes, Nathaniel Titterton, Daniel Armendariz, Luke Segars, Eugene Lemon, Sean Morris, and Josh Paley. Snap! (build your own blocks). In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, page 759, New York, NY, USA, 2013. Association for Computing Machinery.

[HM10] Brian Harvey and Jens Mönig. Bringing “no ceiling” to scratch: Can one language serve kids and computer scientists? *Constructionism*, pages 1–10, 2010.

[HS88] Sandra G. Hart and Lowell E. Staveland. Development of nasa-tlx (task load index): Results of empirical and theoretical research. In Peter A. Hancock and Najmedin Meshkati, editors, *Human Mental Workload*, volume 52 of *Advances in Psychology*, pages 139–183. North-Holland, 1988.

[iso69] ISO R 1028 - Flowchart Symbols for Information Processing. Recommendation, International Organization for Standardization, Geneva, CH, 1969.

[iso73] ISO 1028:1973 - Information processing - Flowchart symbols. Standard, International Organization for Standardization, Geneva, CH, 1973.

[iso85] ISO 5807:1985 - Information processing - Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts. Standard, International Organization for Standardization, Geneva, CH, 1985.

[Jet23a] JetBrains. The State of Developer Ecosystem 2023 - Languages. `https://www.jetbrains.com/lp/devecosystem-2023/languages/`, 2023. Accessed: 2024-02-18.

[Jet23b] JetBrains. The State of Developer Ecosystem 2023 - Team Tools. `https://www.jetbrains.com/lp/devecosystem-2023/team-tools/`, 2023. Accessed: 2024-02-18.

[Man19] Kurt Mandel. Jenkins Plugins: The Good, the Bad and the Ugly. `https://medium.com/@kmadel/jenkins-plugins-the-good-the-bad-and-the-ugly-d7fd0c801a0e`, 2019. Accessed: 2024-02-18.

[MBK+04] John Maloney, Leo Burd, Yasmin Kafai, Natalie Rusk, Brian Silverman, and Mitchel Resnick. Scratch: a sneak preview. In *Proceedings of the Second International Conference on Creating, Connecting and Collaborating through Computing*, pages 104–109, 2004.

[ML07] David J. Malan and Henry H. Leitner. Scratch for budding computer scientists. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '07, page 223–227, New York, NY, USA, 2007. Association for Computing Machinery.

[MPSD19] Steve Mao, Damiano Petrungaro, Zeke Sikelianos, and Lorenzo D'Ianni. Conventional commits. `https://www.conventionalcommits.org/en/v1.0.0/`, 2019. Accessed: 2024-02-18.

[PA10] John Pruitt and Tamara Adlin. *The persona lifecycle: keeping people in mind throughout product design*. Elsevier, 2010.

[Pyta] Python Software Foundation. Python Documentation - Support for type hints. `https://docs.python.org/3/library/typing.html`. Accessed: 2024-02-18.

[Pytb] Python Software Foundation. The Python Standard Library - bdb, Debugger framework. `https://docs.python.org/3/library/bdb.html`. Accessed: 2024-02-18.

[RM17] Partha Pratim Ray and Alok Mishra. A survey on visual programming languages in internet of things. *Scientific Programming*, 2017, 2017.

[SHM+12] Lydia Schneidewind, Stephan Hörold, Cindy Mayas, Heidi Krömker, Sascha Falke, and Tony Pucklitsch. How personas support requirements engineering. In *2012 First International Workshop on Usability and Accessibility Focused Requirements Engineering (UsARE)*, pages 1–5, 2012.

[Smi75] David Canfield Smith. *PYGMALION: A Crative Programming Environment.* PhD thesis, Stanford University, 1975.

[Spa01] Sparck Jones, Karen. A brief informal history of the Computer Laboratory. `https://www.cl.cam.ac.uk/events/EDSAC99/history.html`, 2001. Accessed: 2024-02-18.

[Spe47] Special Committee on Standardization of Therblings, Process Charts, and their Symbols. Operation and flow process charts, 1947. American Society of Mechanical Engineers.

[Sta] Statistisches Bundesamt. Studierende in Mathematik, Informatik, Naturwissenschaft (MINT) und Technik-Fächern. `https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Bildung-Forschung-Kultur/Hochschulen/Tabellen/studierende-mint-faechern.html`. Accessed: 2024-02-18.

[TU a] TU Wien. TU Wien in numbers. `https://www.tuwien.at/en/tu-wien/about-tu-wien/facts-and-figures`. Accessed: 2024-02-18.

[TU b] TU Wien Informatics. Our History. `https://informatics.tuwien.ac.at/history/`. Accessed: 2024-02-18.

[vRWC01] Guido van Rossum, Barry Warsaw, and Alyssa Coghlan. PEP 8 - Style Guide for Python Code. `https://peps.python.org/pep-0008/`, 2001. Accessed: 2024-02-18.

[Wir] Wirtschaftskammer Österreich. Applikationsentwicklung - Coding. `https://www.wko.at/service/bildung-lehre/berufs-und-brancheninfo-applikationsentwicklung-coding.html`. Accessed: 2024-02-18.