WIP: Common Programming Mistakes by Students in an Introductory Embedded Systems Course

Juno L. Robertson, Fana Teffera, and Diane T. Rover
Electrical and Computer Engineering
Iowa State University
Ames, IA USA
{rrcaseyr, fteffera, drover}@iastate.edu

***Abstract*— This work-in-progress innovative practice paper describes programming mistakes commonly made by students in an introductory embedded systems course and explains them using an antipattern framework. Students in introductory programming courses make mistakes in their code resulting in compile-time and run-time errors that may be difficult to understand as novice programmers. Error messages and debugging tools may not offer much help to novice programmers. Recent work in programming education is using language-specific antipatterns to characterize common errors. Antipatterns (mistakes or errors) can be identified in student code, and students then receive tutorial-oriented feedback to explain the error in terms familiar to them from the course. Software tools are automating the process of providing feedback to students about antipatterns found in their code. In this paper, we describe the study being conducted to identify and classify common antipatterns in C code written by students in an introductory embedded systems course. Antipattern data are being collected from the literature, inspection of student code, and instructor, teaching assistant, and student observations and interviews. Relevant and understandable feedback for each antipattern is also being collected and composed. We are also interested in how antipatterns by students evolve over time, the effect of feedback, and how different development environments support feedback to students. This paper summarizes preliminary results of the study, including several embedded C antipatterns, code examples, and feedback. This work has the potential to improve embedded system teaching and learning. The focus on novice programmers highlights the need for specialized educational tools for this group of students and the potential to further personalize individual learning.**

Keywords— embedded systems, debugging, C programming, antipattern

I. Introduction

Novice embedded programmers in electrical and computer engineering fields of study are learning skills and principles they will carry with them into their careers, including coding and debugging skills. They are also learning how to learn, including learning from mistakes. This is complicated by the complexity of an embedded system, which includes both software and hardware, such as a microcontroller, input/output interfaces, sensors, circuits, and other low-level features. In addition, the C programming language is commonly used with embedded systems, and students may be less familiar with C than other languages. Learning through mistakes may be very effective if corrective feedback is provided and paid attention to [1].

In this paper, we describe our early efforts to better understand the C programming mistakes made by students in an introductory embedded systems (ES) course in the Department of Electrical and Computer Engineering at Iowa State University. The sophomore-level course is required in the computer, cybersecurity, and electrical engineering programs, is an elective in software engineering, and is sometimes taken by students later in their program (e.g., EE students). The programming background and skills of the students vary widely, as does the range of mistakes that students make. We are using the concept of an antipattern to study programming mistakes. Various definitions and contexts for antipatterns have appeared in the literature, and we will focus on novice antipatterns, which are errors or mistakes commonly made by novice programmers [2]. Useful feedback for these mistakes is not usually provided by tools used by experienced programmers. Thus we are interested in understanding both common mistakes and useful feedback in support of student learning.

II. Related Work

Antipatterns emerged in software development as a result of finding patterns being used that harmed the software project as a whole [3]. This resulted in identifying and correcting antipatterns, thus improving the health of the overall codebase. Researchers have been studying general programming antipatterns for many years. Antipatterns can take on several different forms. Many compiler errors, logic errors, runtime errors, and even general style issues can be identified and categorized as types of antipatterns. Sixteen style antipatterns are described as representing poor knowledge of programming concepts [4]. A logic antipattern comprised of sequential if statements is portable and relevant to almost any programming language with control structures [5]. There is related work specific to the C language, including common mistakes by novice programmers [6] and a large catalog of C and Python antipatterns [7].

While not explicitly referring to embedded systems, the work on software performance antipatterns for cyber physical systems is closely related [8]. In addition, a list of “bug killing coding rules” for embedded C is effectively a list of patterns, which could be converted into antipatterns [9].

When using antipatterns in education, feedback is especially important. The code critiquing projects in the Department of Computer Science at Michigan Technological University (Michigan Tech) are automating the feedback to students in introductory programming courses using Java, MATLAB, and Python [2] [10]. Feedback specific to style-based issues has been shown to be effective [11]. The debugging C compiler also provides critiques of student code to guide students towards better style and better patterns [12].

III. Antipatterns in Embedded C Programming

A. Context

The introductory ES course (CPRE 288) introduces students to hardware and software aspects of embedded systems using hands-on laboratory activities with a mobile robot and a microcontroller development board. Students learn about microcontrollers, memory, input/output interfaces, embedded programming in C, initialization and configuration of input/output peripherals, input/output methods such as polling and interrupts, debugging, and mobile robots. The first third of the course covers C programming and other foundational concepts and skills. In the middle third, microcontroller peripherals are introduced and used in the lab. During the final third, students implement a project in the lab for an autonomous vehicle application. Early in the course, students are provided with code templates and libraries, and as the course proceeds, students must write more of their own code. The mobile robot is an iRobot Create 2 that is controlled from the external microcontroller board. The microcontroller board interfaces to input/output devices added to the robot, including infrared and ultrasonic sensors and a servo motor. For the lab project, teams of students develop software to move the robot through a test field, detect obstacles, and navigate to a destination (e.g., https://youtu.be/OU5m58bhZ6Y).

The instructor and teaching assistants regularly interact with students and observe their work in several ways, including discussing mistakes and debugging [13]. The course uses a Discord server for asynchronous interaction, and students post messages about errors they encounter with lab activities. Teaching assistants lead lab sections and hold office hours in the lab, thus frequently talking with students about their code and providing feedback. During spring semester 2024, instructional team members took special note of common programming mistakes as they arose to be intentional about understanding the mistakes as antipatterns. An initial list of mistakes is given in Table I. These mistakes appear in student code. For example, code illustrating the mistake "Calling an interrupt handler" is shown in Fig. 5, addressed later in the paper.

In addition to considering antipatterns in the course, a two-semester senior design project was started during fall semester 2023 to create a C code critiquing tool in collaboration with the Michigan Tech group [14]. The code critiquing tool is intended for future use in the introductory ES course to provide feedback to students about antipatterns found in their C code. The team implemented a web application having an interface for the instructor and students. An instructor creates programming assignments and course-specific antipatterns and accesses antipatterns from a database. Students will upload their code and receive feedback. Fig. 1 is a diagram of the web app architecture.

TABLE I. Common Mistakes Observed During the Semester

Two main functions	Code outside of any function body	Not calling init() functions	Calling init() functions repeatedly
Calling an interrupt handler	Full function definition in a header file	Including a C program file (#include)	Copying function call directly from header file
Missing include directives	Initializing variable in extern definition	Initializing members in struct definition	Extern variable that doesn't exist elsewhere
Excessively large arrays	Allocating memory and not freeing memory	Math operations with integers and floats	Using characters not understood by C compiler
Formatting template strings	Mismatch between loop condition and variable values	Mistaking zero character and null character	Not handling null character in strings
Preserving bit fields in registers	Incorrectly using bitwise operations that do nothing	Unintentionally zeroing higher order bits	Incorrectly passing arrays as arguments

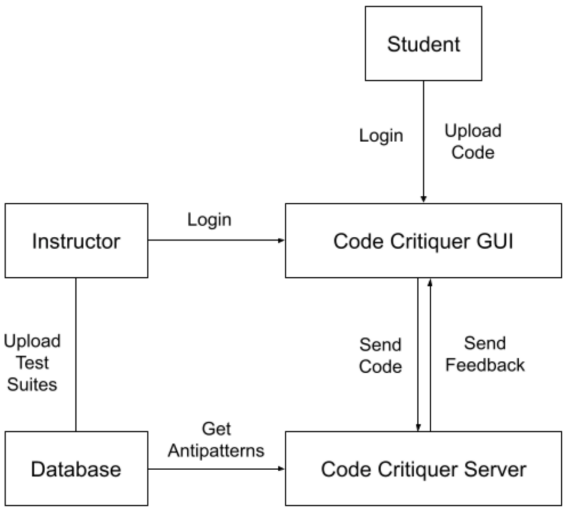

Fig. 1. C code critiquer web app tool.

A sample screen used by the instructor to create or edit antipatterns is shown in Fig. 2. The antipattern shown as an example is an empty loop, which is described with a regular expression.

With further development of this tool, the antipatterns identified by the instructional team will be added to the database and used in the introductory ES course.

B. Methods

A qualitative study involving the teaching assistants in the course was initiated to expand on the list of common mistakes as antipatterns. The initial list was pruned to a subset as a starting point. A preliminary survey was distributed to all of the teaching assistants in the course to gather information that would guide

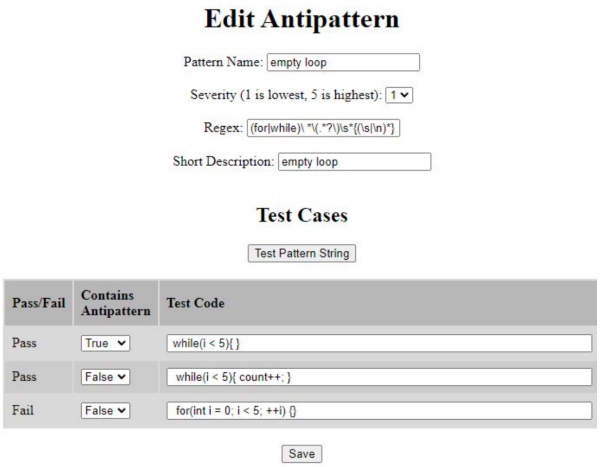

Fig. 2. Instructor sceen to create or edit an antipattern.

subsequent interviews with teaching assistants. The survey was optional. Teaching assistants were invited to participate in interviews with another teaching assistant who was leading the study. The interviews were voluntary and to be conducted either in-person or using Microsoft Teams. They were designed to be conversational and flexible, starting with several basic questions as prompts about specific antipatterns. The interview questions were variations of the following:

- In what situations do you see this antipattern arise?
- How would you classify or categorize this antipattern?
- What steps have you taken to correct this antipattern when you see it?

Seven teaching assistants of varying experience levels participated in these interviews. Audio recordings of the interviews were transcribed. The methods used thus far represent initial coding steps of a thematic analysis methodology [15]. The interview responses were analyzed using Taguette, a tool that supports qualitative data analysis [16]. For example, Fig. 3 is a screenshot from Taguette that shows the tagging process of highlighting text and assigning tags. The panel on the left lists the documents imported into the tool. The panel on the right displays the selected document. Text that has been assigned at least one tag is highlighted in yellow. The dialog box lets the user select tags to apply to the highlighted text. Fig. 4 is a screenshot that shows results after creating and applying tags. The panel on the left lists the current set of tags and, for each, indicates how many highlights each has been applied to. The panel on the right displays all highlights that the selected tag has been applied to, as well as other tags associated with those highlights. Through multiple iterations, these tags become the basis for a codebook, including definitions and sample highlights for each code. This would be followed by grouping codes into initial themes and reviewing the data supporting each theme. These later steps of the thematic analysis are in progress.

C. *Preliminary Results*

The use of two main functions was a common mistake identified by teaching assistants. This is shown in Fig. 4 in the list of tags on the left as the third item. The antipattern is present when a project in the Code Composer Studio integrated development environment for the microcontroller contains multiple main functions. Only one main function is allowed. Comments from TAs about this antipattern associated it with students not reading or understanding compiler error messages, such as symbol redefined. It may have also reflected a general lack of knowledge about the C language compiler requiring

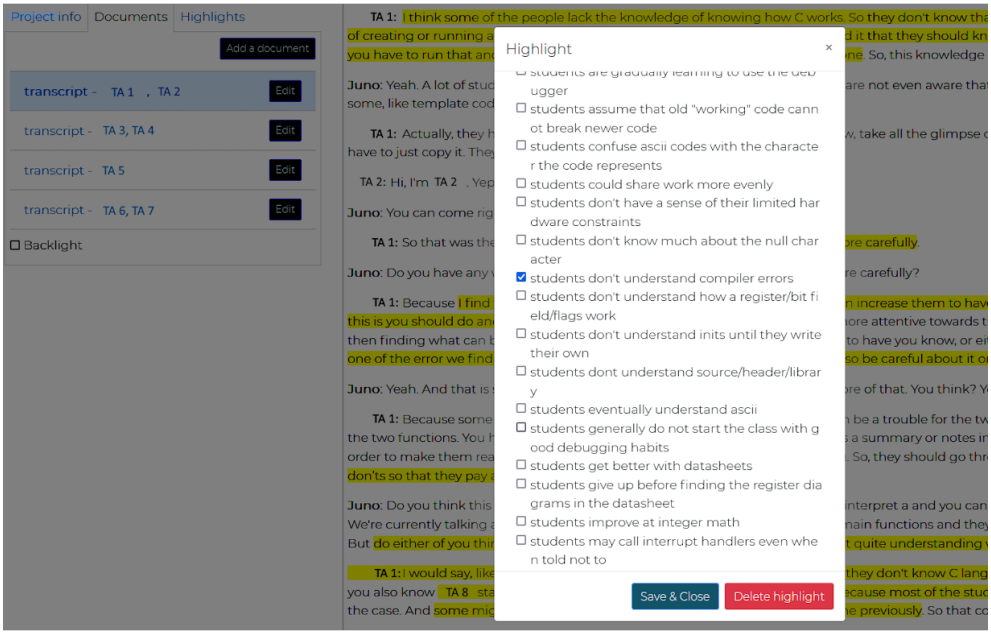

Fig. 3. Screenshot from Taguette tool showing the initial coding process: text data in a document are highlighted and associated with descriptive tags.

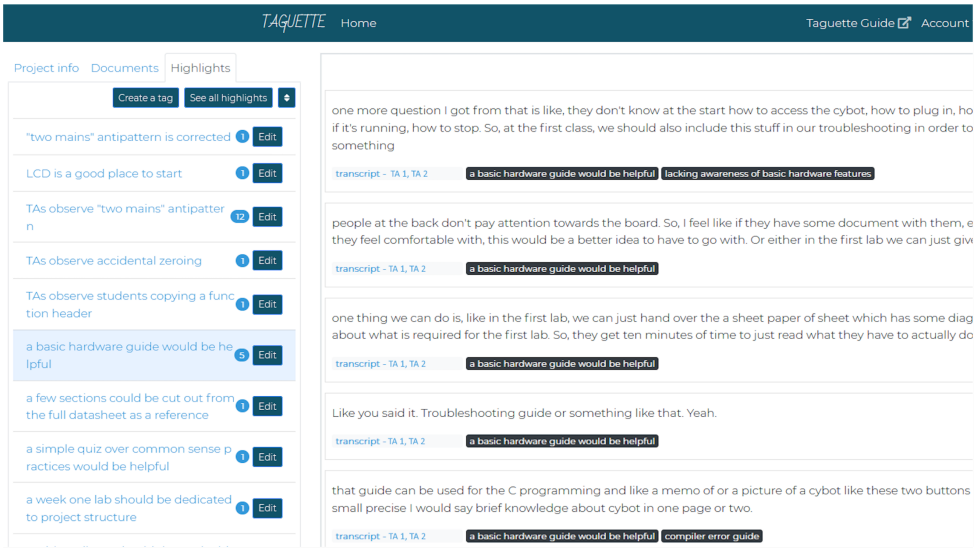

Fig. 4. Screenshot from Taguette tool showing the initial coding process: tag is shown with number and list of highlights it applies to.

unique names in a program. It may also show a lack of knowledge or attention to how C projects are structured. They may have copied some previous code into a new project and not double-checked the files. However, once students receive feedback about this antipattern, they are less likely to make this same mistake again.

Calling the interrupt handler directly by students in their code is an antipattern. The interrupt handler function, also known as an interrupt service routine, is meant to be invoked automatically by the interrupt system when a trigger event occurs, such as receiving a character or a sensor input. Fig. 5 shows a code example in which an interrupt handler function is called inside another function in the student's code. The interrupt handler function must not be called like a regular function is called. It is a special function supported by the microcontroller hardware. Fig. 6 shows the same scenario, but in this code example, the interrupt handler is not called. It will be executed automatically by the hardware when the trigger occurs, and it will update the values of variables read by the student's code (i.e., status flags in Fig. 6). This antipattern is more likely to be found in embedded systems and in code by novice embedded programmers. Students may be confused by the terminology and not realize that interrupts behave like events in software. They don't yet understand that hardware is designed to support program execution in various ways, including automatically executing interrupt handlers when interrupt events occur.

```
int_32t ping_get_width_cycles(){
      ping_send();

      interrupt_handler();

      return time_up - time_down;
}
```

Fig. 5. Code fragment in which the interrupt handler function is erroneously called in student code.

Another interesting antipattern for embedded systems is declaring excessively large arrays, which means that arrays use more physical memory on the microcontroller than is required for the program and/or than is available on the microcontroller chip. This often results in runtime errors that are very difficult to debug because of unpredictable behavior or failure. A typical microcontroller has a limited amount of memory for program code and data. However, this is often an entirely new constraint for students to consider when they have been writing programs for desktop or server computers that have very large amounts of memory and sophisticated memory management systems.

The large array antipattern is directly related to physical aspects of the embedded system, much like software performance antipatterns in cyber physical systems are not only dependent on software [8]. This type of antipattern, specifically, hardware-software antipatterns for embedded systems, appears to be new and is an area for future work. Several of the mistakes in Table I, such as bitwise operations on input/output registers, are of this type.

```
int_32t ping_get_width_cycles(){
      ping_send();
      while (!flag_sent | !flag_up |
            !flag_down) {} //wait for flags
      flag_sent = 0;
      flag_up = 0;
      flag_down = 0;
      return time_up - time_down;
}
```

Fig. 6. Code fragment in which student code uses status flags updated by the interrupt handler instead of calling it.

IV. Discussion and Future Work

While the study is ongoing, we have made some general observations from the interview data, instructional team discussions, and interactions with students.

Students often have the knowledge needed to fix mistakes in their code. However, they may hesitate and be uncertain about what action to take. Sometimes a solution becomes apparent by scrolling up to read more of the compiler error messages in the CCS IDE. Sometimes looking up the symbols in question provides enough information. Sometimes it may be as simple as a spelling mistake, maybe even shown in the error message itself. It's important to help students gain the confidence to investigate these errors instead of being intimidated by them. Feedback that helps students take the initiative may be especially beneficial. These behaviors are associated with a debugging mindset [13].

Even if students have knowledge or interest gaps, there are ways for teaching assistants to relate mistakes back to more familiar concepts. Bridging these gaps with feedback and encouragement may lead to greater initiative and engagement in the course. In fact, students who struggle with mistakes earlier in the course often have shown improvement later in the course. Thus, the process of making mistakes and receiving feedback to correct them offers opportunities for students to see progress in the course.

When exploring the challenges students face as novice embedded C programmers and the learning strategies involved in C programming and embedded systems, the transition from programming languages such as Java and Python to C is one of the hurdles. These students often need assistance with fundamental C programming concepts such as data types, explicit memory management, low-level coding practices, pointer usage, and bitwise operations. Students must adapt their mental model from previously learned programming frameworks based on higher-level system and data abstractions. In addition, understanding the capabilities and limitations of embedded systems is a critical area of development for novice embedded programmers. When students struggle with embedded system limitations (e.g., memory constraints), there is an opportunity to improve not only their programming but their understanding of the system as a whole.

In addition to future work in the development of hardware-software antipatterns, we are interested in the advancements in artificial intelligence (AI) for code generation, debugging with AI, coding assistants, intelligent tutors, and personalized learning. With a recent announcement by GitHub about Copilot Workspace support for using natural language to code, this further begs the question of what's possible for novice programmers [17]. For now, consider this quote, "You still need to know the underlying language well enough to be able to debug it." [18] Thus, future work will continue to explore these advancements and how to leverage them.

Acknowledgment

The authors acknowledge the contributions of the teaching assistants for the embedded systems course and the members of the senior design team.

References

[1] J. Metcalfe, "Learning from Errors," Annual Review of Psychology, vol. 68, 2017, pp. 465-489, https://doi.org/10.1146/annurev-psych-010416-044022 (https://www.annualreviews.org/content/journals/10.1146/annurev-psych-010416-044022)

[2] L. C. Ureel II, "Critiquing Antipatterns In Novice Code", Open Access Dissertation, Michigan Technological University, 2020. https://doi.org/10.37099/mtu.dc.etdr/1158

[3] W. H. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray, *Antipatterns: Refactoring Software, Architectures, and Projects in Crisis*, 1998, John Wiley & Sons, Inc.

[4] G. De Ruvo, E. Tempero, A. Luxton-Reilly, G. B. Rowe, and N. Giacaman, "Understanding semantic style by analysing student code," Proceedings of the 20th Australasian Computing Education Conference, 2018, pp 73–82.

[5] S. Nurollahian, M. Hooper, A. Salazar, and E. Wiese, "Use of an anti-pattern in cs2: Sequential if statements with exclusive conditions," Proceedings of the 54th ACM Technical Symposium on Computer Science Education, SIGCSE 2023, 2023, pp 542–548.

[6] T. Delev and D. Gjorgjevikj, "Static analysis of source code written by novice programmers," 2017 IEEE Global Engineering Education Conference (EDUCON), 2017, pp 825–830.

[7] Y. Bosse, I. S. Wiese, M. A. G. Silva, N. Lago, L. de Oliveira Brandao, D. Redmiles, F. Kon, and M. A. Gerosa, "Catalogs of C and Python Antipatterns by CS1 Students," 2021, https://arxiv.org/abs/2104.12542.

[8] C. U. Smith, "Software Performance Antipatterns in Cyber-Physical Systems," Proceedings of 2020 International Conference on Performance Engineering, 2020. https://doi.org/10.1145/3358960.3379138 https://research.spec.org/icpe_proceedings/2020/proceedings/p173.pdf

[9] M. Barr, "Bug-killing coding standard rules for embedded C," Barr Group, May 4, 2016, https://barrgroup.com/blog/bug-killing-coding-standard-rules-embedded-c

[10] L. C. Ureel II and C. Wallace, "Automated critique of early programming antipatterns," Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE '19, 2019, pp 738–744.

[11] E. S. Wiese, M. Yen, A. Chen, L. A. Santos, and A. Fox, "Teaching students to recognize and implement good coding style," Proceedings of the Fourth ACM Conference on Learning @ Scale, 2017.

[12] A. Taylor, J. Renzella, and A. Vassar, "Foundations first: Improving c's viability in introductory programming courses with the debugging c compiler," Proceedings of the 54th ACM Technical Symposium on Computer Science Education, SIGCSE 2023, 2023, pp 346–352.

[13] H. Duwe, D. T. Rover, P. H. Jones, N. D. Fila and M. Mina, "Defining and Supporting a Debugging Mindset in Computer Engineering Courses," 2022 IEEE Frontiers in Education Conference (FIE), Uppsala, Sweden, 2022, pp. 1-9, doi: 10.1109/FIE56618.2022.9962605.

[14] N. Carber, C. Cook, B. Ford, Em Huisinga, S. Matt, and C. Robison, "Code Critiquer System for the C Language," Project Website, ECpE Senior Design, Electrical and Computer Engineering, Iowa State University, https://sdmay24-34.sd.ece.iastate.edu/

[15] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, 3(2), 2006, pp. 77–101.

[16] R. Rampin and V. Rampin, "Taguette: open-source qualitative data analysis," *Journal of Open Source Software*, 6(68), 2021, https://doi.org/10.21105/joss.03522 (https://www.taguette.org)

[17] T. Dohmke, GitHub Copilot Workspace: Welcome to the Copilot-native developer environment, GitHub Blog, April 29, 2024, https://github.blog/2024-04-29-github-copilot-workspace/

[18] S. Hymel, Twitter/X post, May 13, 2024, https://twitter.com/ShawnHymel/status/1790025445942927716?s=19