Leiden University

ICT in Business and the Public Sector

Hedy programming language
introducing the Gradual Feedback Model (GFM)

Name: T.B. Bakker

Date: 22/09/2021

1st supervisor: Dr.ir. F.F.J. Hermans
2nd supervisor: N. van Weeren MSc

MASTER THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

In this Thesis, a data analysis is conducted on over 1 million programs written in Hedy to better understand the struggles novice programmers face when learning to program. Hedy is a *gradual* programming language aimed at novice programmer programming, which introduces new programming concepts gradually to reduce cognitive load while learning. We find a pattern of re-submitting identical faulty code, not reading the error messages and high drop-out rates. Statistics that verify the difficulty novice programmers encounter when learning a programming language, a problem that has been studied and verified by similar research. While Hedy is developed to make learning to program easier for starting programmers, we argue that improvements can still be made.

We propose a new model of returning error messages with the aim to solve the found patterns and difficulties of novice programmers: The *Gradual Feedback Model*. The model proposes a different way of returning errors, making them dependent on the behaviour and progress of the programmer. The model keeps track of a *feedback level* and returns a corresponding error message. Varying in levels of helpfulness for solving the error at hand. Through *enforced error reading* and *duplicate faulty code prevention* novice programmers are stimulated to read and interact with the error messages. It is implemented within the Hedy web environment and tested through A/B testing. While the focus is on Hedy the model itself is language-independent and can be implemented in any text-based programming language such as Python.

We find a decrease in error rate by users using the *Gradual Feedback Model*. Through statistical analysis we conclude on a definitive statistical significance of the model. No decrease in drop-out rate is found. User usefulness is gathered through yes/no questions on the model interaction. A decreasing usefulness rate is found where users rate the model as less useful after multiple interactions. The usefulness as rated by users differs greatly for the different levels of feedback and does provide enough insights for future improvements of the model. Also a low interaction rate with the model is found, whereas less then 40% of the users presented with the model actually interact with the feedback. The cause of this low rate is unknown and an interesting starting point for future research. Further research through observational studies should be conducted to get more insights in the usefulness of the model as well as enabling the improvement of the different feedback levels.

Acknowledgements

I would like to thank Felienne for supervising and helping with this research, the feedback throughout the year as well as the nice talks (although mostly online) we had on Hedy. I would also like thank Federico for all his help on implementing the model, working together on the endless merging errors and his work on the implementation of A/B testing within the Hedy environment. A feature that was essential for the testing of the GFM implementation. And thanks to Niels for being second supervisor and giving a lot of useful feedback and help in the last stage of the process. Lastly, I would like to thank the PERL research group, for which the weekly group meeting was a nice moment of socializing, structure and feedback in this unusual time of graduating.

Contents

1 Introduction **6**

2 Programming: Mistakes we make **7**
2.1 Novice programmers 7

3 Gradual programming language: Hedy **10**
3.1 Level structure 11

4 A Data Analysis: Hedy **12**
4.1 The dataset 14
4.2 Concept usage 16
4.3 Language distribution 16
4.4 Program length and distribution 18
4.5 Error analysis 20
4.6 Drop-out rate 22
4.7 Identical submission analysis 23
4.8 Time to Change 24
4.9 Steps to Success 26

5 Enhancing the Error Message **27**
5.1 Literature review 28
5.1.1 Conclusion 30
5.2 The Gradual Feedback Model (GFM) 31
5.2.1 Overview 31
5.2.2 Getting the feedback 33
5.2.3 Design choices 33
5.2.4 Feedback as a debugging strategy 34

6 Implementation **35**
6.1 Overview 35
6.2 Error handling 35
6.2.1 Current Hedy error handling 35
6.2.2 GFM implementation Hedy error handling 36
6.3 Finding similar (correct) code 38
6.3.1 Pre-processing the data 38
6.3.2 Pre-processing the submitted code 40
6.3.3 Limitations 41
6.4 The front-end 42
6.4.1 Enforce error reading 43
6.4.2 Retrieving user feedback 43
6.4.3 Preventing identical faulty code 44
6.5 Expanding the log 45
6.6 Limitations & Challenges 46

7 Results **47**
7.1 The data 47
7.2 Comparison Analysis 49
7.2.1 Error rate 49

7.2.2 Drop-out rate . . . 50
7.2.3 Steps to success . . . 51
7.3 Usefulness analysis . . . 53
7.3.1 GFM interaction rate . . . 53
7.3.2 Feedback usefulness . . . 57
7.3.3 Copying analysis . . . 59

8 Discussion **61**
8.1 Results . . . 61
8.2 Limitations . . . 61

9 Future Work **62**
9.1 Improving the implementation . . . 62
9.2 Increasing usefulness . . . 62
9.3 Language-independent implementation . . . 62

10 Conclusion **63**

11 Appendix A: GFM Hedy Messages **66**

1 Introduction

For a long time, computer programming has been seen as an exclusive STEM skill. Highly correlated with the ability to solve problems systematically. For which, it was seen as similar to solving problems in mathematics. However, programming is more than solving problems. Programming a solution is not only about solving the problem, but beforehand a long learning curve of syntax, errors, exceptions and other programming specific context has to be learned. Reading, understanding and solving error messages are a skill that isn't correlated with the aspects of STEM skills and shouldn't be approached like a mistake in a mathematical equation. So why is the learning of programming still approached like an exclusive STEM skill? [17]

In this research, we take an in-depth look at the usefulness of error messages in programming languages. With the aim to better understand the bottlenecks of novice programmers when learning to program. First, an introduction is given in the research already done on general mistakes made by programmers, especially novice programmers. Then we look at Hedy, a newly introduced gradual programming language, with the aim to solve (some of) these bottlenecks and take a different approach to the learning of programming. Hedy is developed with the novice programmer in mind, bridging the gap from no programming experience towards Python programming by reducing cognitive load on the learning journey. It is structured in levels, whereas in each level new keywords and programming concepts are introduced. A data analysis is performed on the Hedy programs dataset, containing over 1 million programs. This to get insights in the mistakes made within the Hedy programming language and the ways users respond to and interact with error messages.

Then we look at related work on improving error messages and introduce a new way of given error feedback: the *Gradual Feedback Model* (GFM). A model for which the interpreter or compiler keeps track of the mistakes of the user and alters the message on their understanding. Giving more explanatory errors, general programming tips and code suggestions by consecutive mistakes. Through the use of a *Feedback Level*, the needs on error interaction are scaled to the user. Identical (wrong) code prevention and *enforced error reading* are implemented by keeping track of the user's interpreter interaction. The model is implemented within the Hedy web environment, but the concept itself is largely language-independent and can be implemented in any text-based programming language, such as *Python*.

Further on, the implementation of the model is discussed. Comparing the current error handling with the new GFM approach. Giving a code explanation of the implementation done. An A/B test is performed on the Hedy environment with and without the GFM implementation. The results are divided into two analysis: A *comparison analysis* and a *usefulness analysis*. In the first one, an analysis is done comparing general coding statistics with and without the GFM implementation. Such as *error rate* and *drop-out rate*. In the usefulness analysis, an analysis is performed on the user feedback. Analysing the *interaction rate* and *usefulness rate* of the model in general as well as the specific levels of feedback. This is followed a conclusion on the performance of the model. The limitations of the model and the implementation are discussed as well. Lastly, this Thesis finishes with a suggestion for future work and a general conclusion of the work done.

2 Programming: Mistakes we make

In this section, we look at programming mistakes made by novice programmers, independent of programming language. The focus of this section is on starting programmers due to this being the target group of the Hedy programming language as well as this research. We take a closer look at these mistakes to get a better understanding of the learning bottlenecks. Learning to program is like learning a new language, with one big difference: the programming language doesn't try to understand you. Where in a spoken conversation people might ask for clarification when something is unclear, programming languages don't: they simply respond to your commands. This is an important misconception that arisen in the earliest steps of programming. As stated by R. Pea: "*First, we need to be aware of the pervasiveness of programming misunderstandings that arise from the tacit applications of human conversational metaphor to programming.*" [19]

It is not the case that novice programmers *think* that the computer has an actual mind of his own. The understanding of the concept that computers are unable to think for themselves seems to be understood well. However, novice programmers tend to assume some type of intelligence or *co-operation* of the computer. Approaching it more like a natural language conversation than a command-like conversation. This and other well-known misconception are discussed in this section and will help us better understand the struggles of novice programmers that have led to the structure of Hedy. Much research has been done in analysing the mistakes of novice programmers in a larger scale such as [2][5][6] as well as more general mistakes of *professional* programmers that might introduce vulnerabilities into programs [4].

2.1 Novice programmers

In this subsection, we focus on mistakes made by novice programmers. We only look at *language independent* errors; namely, the errors that are possible to occur in any text-based programming language, such as syntax errors and logic errors. For example, memory management errors would not occur in languages that automatically handle memory management, such as Java and Python. While not all syntax errors might be language-independent, we focus on the ones that are. Other errors found in research are left out of scope. Monika Kaczorowska researched programming mistakes made by first and second year IT students programming in the C++ language. A total of 602 programs is collected from 43 students. The same 43 students account for the data on the first as well as the second year students. She found that the mistakes can be classified within three categories: syntax errors, memory management errors and logic errors. [14] A (abbreviated) overview of the most frequent errors found is shown in Table 1. Note: a *shadowing variable* is a variable that is declared outside a function or loop and then again inside a function or loop. This way, the firstly declared variable is and can never be used within the scope of the function or loop. They found that more mistakes are made by first year students than second year students. And the most problematic errors seem to be related to *memory management*, whereas second year students also had difficulty with *pointers*. It should be noted that both these errors would not apply for programming in either Python or Hedy because these constructs are either handled automatically by the interpreter (memory) or are not available (pointers).

Syntax error	Logic error
Invalid number of parameters passed to function	Shadowing variable
Variable names containing the space	Incorrect condition in loop
No brackets or odd number of brackets	Incorrect initial value of variable
Visibility of variables in switch case section	Uninitialised variable

Table 1: General programming errors found by 1st and 2nd year IT students [14]

When we take a closer look at the table, we notice a pattern: Some common mistakes are not a problem of misconceptions or misunderstanding of the logic behind programming. We can classify all syntax errors as non-logic errors; occurring due to structures non-relevant for the logic behind the program. In the case of the logic errors, we can also argue that the errors of either a *Shadow* or *Uninitialised* variable are due to syntax barriers and not due to logic. It might be better to classify them as semantic errors (correct syntax but incorrect use). However, for unknown reasons semantic errors were not a category in the research and possible semantic errors are classified as syntax errors.

Altadmri and Brown analysed 37 million compilations from over 250.000 students programming in Java. The data stemmed from the Blackbox dataset, which was gathered from users of the BlueJ Java IDE. An IDE specifically developed for novice programmers. [16] They generalize the mistakes made into 18 mistakes, splitting into three categories: *misunderstanding syntax*, *type errors* and *other semantic errors*. An example of a semantic error in Java is calling a non-void function but not using or storing the return value. They found that *syntax errors* such as *mismatched brackets* and *quotations* are the most frequent ones amongst novice programmers, but also the quickest to be fixed. One interesting finding is that the frequency of *semantic errors* increases over time, while the frequency of *syntax errors* decreases. With which they suggest that semantic errors are a more serious challenge to solve for novice programmers than syntax errors.

While less common, research in errors made by novice programmers in Python has also been done. A. Junior et al. focus on the impact of making mistakes on the programming abilities is children age 14 till 16. By manually checking code submissions they find 31 mistakes which could be seen as a pattern, 15 of these are found when analysing the dataset using a code analyser. It should be noted that Python 2.7 is used, which is outdated and has differences with the currently used Python 3. They found the most common mistakes are: *unused variables*, *redundant float conversion* and *wrong input reading*. Whereas, we can argue that the *unused variable* mistake is not actually an error because it doesn't result in one. However, it is a programming mistake and creates unnecessary cluttered code. [13]

One large research in Python errors has been done by Tobias Kohn. He states that Python has for long stated out of the discussion in the field of compiler error messages. Which can be confirmed by the relatively low amount of research in Python errors. He analysed a total of 4091 error instances, of which 1233 are classified as minor. It is classified as minor if it could be solved by a simple edit; for example, replacing one character or swapping two characters. In this research context the scope of mistakes is on the *SyntaxError*, *IndentationError*, *NameError*, and *TypeError*. The most common errors can be found in Table 2. He found that a considerable part of errors are due to minor mistakes and easy to fix. Stating that the reason why enhancing error messages often appears to be inefficient is due to error being easy to fix or not being captured within the error message at all.

Most common errors
Name Error: Cannot Find Name
Wrong/Inconsistent Indentation
TypeError
Missing Comma or Operator
Missing/Mismatched Brackets

Table 2: Most common Python errors found by Tobias Kohn [15]

Lastly, a more recent research in Python errors by novice programmers has been performed by Rebecca Smit and Scott Rixner. [22] Data was gathered from an online Python course for which the students coded in *CodeSkulptor*, a browser-based IDE. The dataset contains over 330.000 implementations from over 95.000 *development chains*. Interesting in their research is their focus on other facets of the error landscape than the frequency distribution of errors, the aspect that is mostly researched in other found work. They look at aspects such as the duration and evolution of errors. They found that *TypeErrors* are most likely to occur as well as re-occur within the development chain. While *SyntaxErrors* where common as well, but did occur on smaller subsets of the chain. Interestingly, *IndexErrors* and *KeyErrors* do occur less but appear longer within a chain. Implying that students have more difficulty solving these errors.

When leaving out the language-dependent mistakes and errors, a pattern can be found over the different research analysed on programming mistakes. [13][14][15][16][22] Several mistakes that can be found across all research are trying to use a *non-existing variable*, having issues with *mismatching brackets* as well as wrongly *initialising a variable*. Either through a wrong type or starting value. Mistakes when, thinking about the different programming languages, are difficult to prevent. But, for unknown reasons, novice programmers seem to struggle with these. Other interesting findings are the difference in difficulty of the type of errors. While some types are found and solved frequently, others occur less but are harder to solve. Insights which are useful when implementing error messages. Extra focus and attention should aim at these *hard to solve* errors to better facilitate novice programmers on their error solving journey.

3 Gradual programming language: Hedy

In this section, the *gradual programming language* Hedy is discussed. Hedy is a text-based programming language developed by Hermans in 2020. [12] The unique aspect of Hedy is the syntax development: the language gets gradually more complex and complete while the student is learning. Starting at level 1 with an option to *print* text and gradually introducing new concepts and more python-like syntax structures along with the levels. Limiting the *cognitive load* on students that *traditional* programming languages introduce due to the large amount of syntax to be remembered at once as well as the programming concepts themselves. [23] It should be noticed that the target group of the programming language are *novice* programmers, especially children. While the language can feel like a *tutorial* due to the way you can advance a level to be able to use more complex features: it is still a Turing-complete programming language which can be used to create any program in mind.

The syntax of Hedy is gradually moving towards the Python syntax: together with the learners. For which, it might be best to classify the language as introduction to programming in *Simplified Python Syntax*. Leaving out aspects that increase the cognitive load but don't contribute to the actual learning of programming. It is built upon Python, and the Hedy code is parsed to Python before being executed. One important note on this level structure is that code that executes successfully on one level might give an error on another level. This due to the changing syntax of core aspects, such as print. For example, in level 1 a text is printed using the *print* keyword combined with the string whereas from level 3 and on the string should be placed between quotation marks. See the examples below:

Printing a string in level 1:

```
print Hello world!
```

Printing a string in level 3:

```
print 'Hello world!'
```

Hedy is open-source and available through GitHub. [10] It can be either run locally or on the Hedy web environment at `http://hedycode.com/`. At the web environment, users are able to get familiar with programming concepts through *Adventure Mode*. Short exercises that get more complex and functional throughout the levels, together with gradually expanding syntax. In addition, users can create an account and store created programs online. An visual overview of the web environment can be found in Figure 1.

One can understand that this gradual concept might complicate programming when executing code within the levels in a non-linear structure. Because correctly executing code in one level might run into errors at another level. Therefore, more experienced programmers should keep the learning aspect in mind, trying to place themselves in the position of a novice programmer who starts at level 1. Programmers who want to program as much functionality as possible should read the syntax and start programming in the highest available level, or start programming in Python.

3.1 Level structure

The Hedy language is originally structured into a total of 13 levels. However, currently 22 levels have been implemented due to ongoing development on the language and other parallel research on expanding the levels. In this subsection, a short introduction of each level is given to create a better understanding of the language. Here, and in the rest of this research, we focus on the first 13 levels. This decision is made due to the original Hedy paper proposing and describing the first 13 levels. For these levels the structure is known, and they are no longer subject to change due to ongoing research. A more in-depth explanation of the level structure of Hedy can be found in either the original paper by Hermans or the Hedy website. [12][11]

Level	Main concept(s) introduced
1	Printing and input
2	Variable assignment
3	Quotation marks, types
4	If and else-statements
5	Repetition
6	Calculations
7	Code blocks
8	For-loops
9	Colons
10	Nested repetition / selection
11	Round brackets
12	Rectangular brackets
13	Replacing is with = or ==

Table 3: Hedy level structure for the first 13 levels [12]

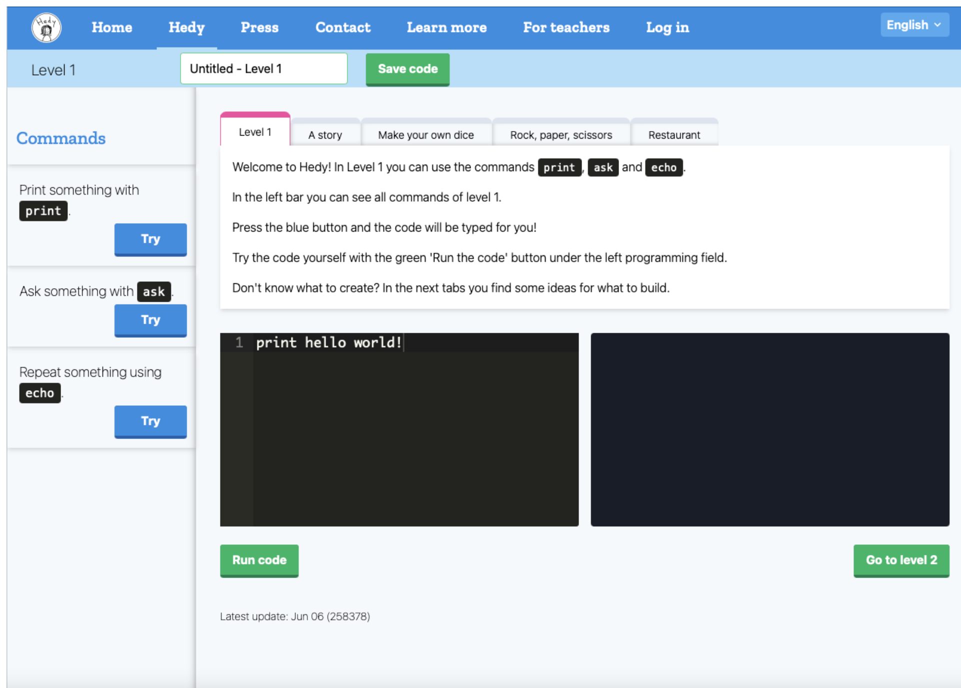

Figure 1: Hedy web environment overview

4 A Data Analysis: Hedy

In this section the Hedy dataset is introduced, discussed and analysed. The dataset contains all Hedy programs executed through the Hedy web environment, as well as additional values which will be discussed more in-depth further on. To get a better understanding of the mistakes made by novice programmers as well as common mistakes made within Hedy we analyse the currently available data. Each program run through the Hedy web environment is logged and stored in a dataset. This is also the case for submissions that results in an error or don't contain any code at all. At the moment of writing, a total of $1,209,123$ programs is created in level 1 through level 13. Each logging, row in the dataset, consists of 14 values, for which an overview can be found in Table 4. For each value, the data type and a short explanation of the value is given as well. Throughout the development of Hedy the structure and amount of values being logged has changed multiple times. Therefore, not all variables have been present since the start of the logging process. A random row from the dataset can be found in Table 5. An explanation of relevance of each value is given in this section.

The *session* value contains a unique hash-values to keep track of the current user. Multiple code submissions by the same user in the same *session* will be stored with the same session value. The *date* value is a timestamp of the code submission. The *level* value is the Hedy level the code is submitted in. This is important because the Hedy syntax differs for each level. The most relevant information is stored in the *code* value, which contains the code of the submission. The *server_error* value contains the server error *if* any error occurred. Otherwise, this value will be -. Any error that occurs during the parsing will result in a server error. So automatically, all programming errors will be logged in this value as well. The *version* value contains the version ID of the Hedy environment in which the code is submitted, as well as the date of version deployment. The *submission_id* value is also a unique hash-value, however where the session value can occur multiple times, this value is unique for each submission. Which can be seen as the identifier of the logging, or *primary key* from a database perspective. The *lang* value contains the currently selected language in the Hedy web environment. Note that the programming language itself is language-independent, but the explanation, example code and error message is translated for each language. It is stored as the abbreviated language of choice, e.g. *nl* or *de*. The *demo* value is an indication if example code is run, which can be selected from the interface choosing one of the examples. Similar to the *start* value, which stores if the starting code of the current level is submitted.

In this section, the data is used to get insight in the mistakes made by programmers. First, a general analysis is performed, visualising general statistics such as programs and error percentages per level. We then look at the program distributions, such as the *word count* and *concept usage*. Followed by a *drop-out* analysis on each level. We assume that dropping out is an important indicator of insufficient help on solving the current errors. Then, an *identical code* analysis is done. Analysing the patterns of consecutive identical (faulty) code submissions. Finally, we look at the *steps till solve*, analysing the amount of faulty submissions before a correct one is made. The dataset is a .csv-file and all analysis is performed using Python 3.8.3. Through the use of the *Pandas* library, the dataset is pre-processed and visualized with either the *Matplotlib* or *Seaborn* library. All code is run locally using *Jupyter Notebook* on macOS Big Sur 11.4 and available on request.

Name	**Data type**	**Value explanation**
Session	String	Unique string for each session
Date	Date	Date and time of code submission
Level	Integer	Level of code submission
Code	String	The actual Hedy code
Server_error	String	Server error if occurred, otherwise a "-"
Client_error	String	Client error if occurred, otherwise a "-"
Version	String	Version of Hedy on which the code is executed
Submission_id	String	Unique ID of each code submission
Lang	String	Language of Hedy web environment
Email	String	User email address if logged in, otherwise a "-"
Username	String	User username if logged in, otherwise a "-"
Is_test	Boolean	*True* if logged on test environment, otherwise *None*
Demo	Boolean	If the submitted code is an example one
Start	Boolean	If the submitted code is a starting one

Table 4: Features logged and stored at each code submission

Variable name	**Value**
Session	64b2ec0ad3944199bea214d02d028f17
Date	2021-03-08 14:14:19.820292
Level	2
Code	player1 is rock, paper, scissor \n print Player1...
Server_error	-
Client_error	-
Version	Mar 04 (74f3a0)
Submission_id	6046313c121bf907dd9926fd
Lang	en
Email	-
Username	None
Is_test	-
Demo	False
Start	False

Table 5: Example of a submission logging through the Hedy web environment

4.1 The dataset

The Hedy dataset consists of $1,209,123$ code submissions by $339,376$ unique sessions from level 1 through level 13. As stated earlier, higher Hedy levels are out-of-scope in this research. It should be noted that this is not the same as the unique amount of users. A new session is created each time a user enters the website. When a user uses the Hedy web environment on multiple devices or returns after the session has expired these are all counted as unique sessions. The username value has been implemented recently and creating an account is not mandatory. Through this structure the exact amount of unique users is unknown. The structure of the dataset is changed multiple times, where for example the *server_error* value is either a *None*, - or the actual error message. The error messages are returned to the user in the supported language of choice in which the error messages are stored in the dataset as well. Due to not containing any general labelling complicating the process of filtering and counting specific error messages. The error messages are stored in one of the (at the moment of writing) 9 supported languages. The *client_error* values suffer from the same complications as the *server_error* values. Client errors are out-of-scope of this research due to them not being directly related to the programming itself. They can occur due to a mistake when parsing, Hedy code is parsed to incorrect Python code, or any other unexpected error on the client-side of the application. For completeness, they are shortly mentioned in the following subsection.

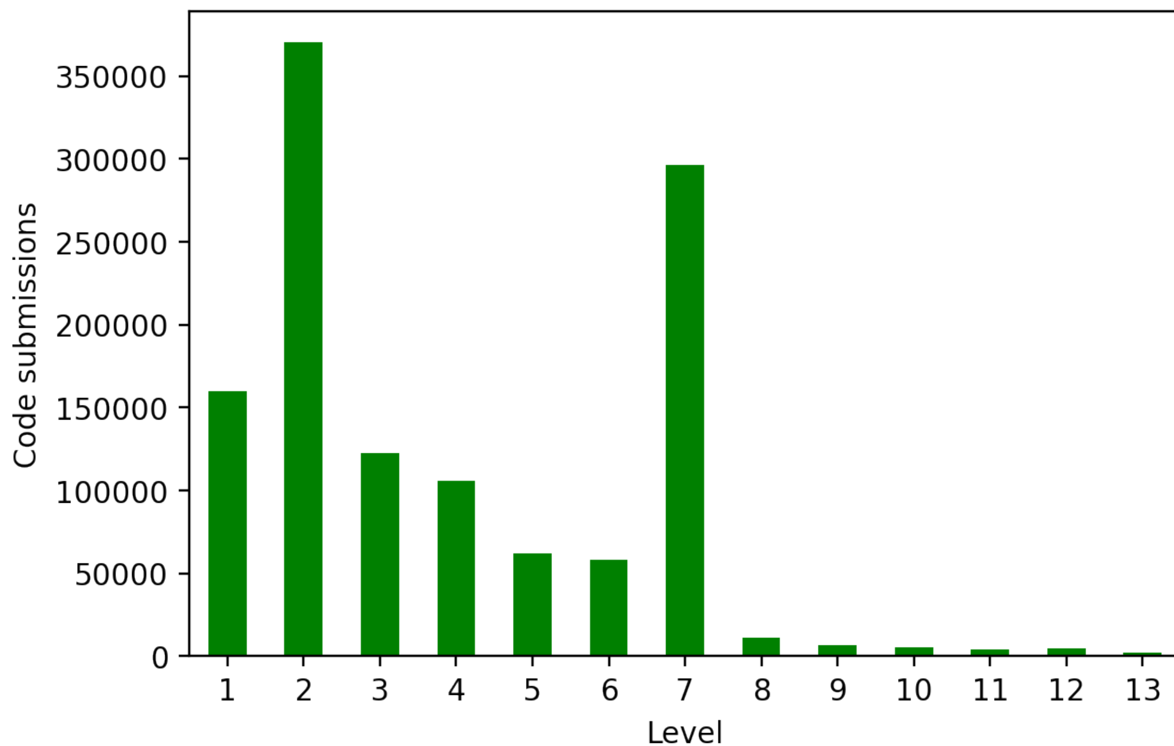

Figure 2: Hedy programs count per level

In Figure 2 an overview is visualized of the total amount of code submissions per level. One would expect for level 1 to have the largest amount of programs as this is the starting point of Hedy. While this expected diminishing pattern is the case for most consecutive levels, two outliers can be found. Level 2 as well as Level 7 have an unexpected high amount of code submissions. The amount of submissions for level 2 can be explained by the introduction of the *random* keyword in this level. Enabling users to get different results with the same code and inviting them to submit the same code multiple times. The large amount of programs for level 7 can be explained due to being the last deployed level for a long time. Expecting that users who were interested in the most functionality possible at that point would execute their code on level 7.

An overview of the deployment time between levels can be found in Table 6. One can notice that the first 7 levels of Hedy where deployed relatively fast after each other, while the time between these and Level 8 as well as the time between level 8 and 9 is significant. Afterwards, the following levels where deployment quickly after each other as well. The levels developed and deployed after level 13 are out-of-scope of this research and therefore not included in this table. It should be noted that the deployment dates are based on the dataset and might not be entirely accurate. For example, when a level is tested in development and logged while not being deployed, in this case the first logged date is counted as the deployment date. This might be the case, for example, for level 3 for which a logging is found earlier than for either level 1 or 2.

Level(s)	First log date	Hedy Version	Days since last level
1, 2	19-03-2020	739f13	-
3	18-03-2020	ef7451	-
4	19-03-2020	e2913d	1 day
5	23-03-2020	2d4993	4 days
6	31-03-2020	82ebb2	8 days
7	05-04-2020	76d4af	5 days
8	13-10-2020	ae1b38	191 days
9	01-04-2021	da179e	170 days
10	06-04-2021	50aafc	5 days
11, 12, 13	16-04-2021	e06fe9	10 days

Table 6: Deployment overview of first 13 Hedy levels

4.2 Concept usage

In this subsection, we look at the Hedy concept usage over the different levels. The Hedy programming language introduces new concepts and coding structures throughout the levels. The normalized concepts used per level can be found in Figure 3. The percentages in this figure show the percentage of programs that contain these concepts. For example, it is clear that the *echo* keyword is only allowed in level 1 as it is not used further on. Similarly, one can see that from level 2 and onward, the *print* keyword is used in nearly all programs. No filter is applied for keyword usage within strings. For example, when a string is printed containing a keyword, this is still counted a use of concept. This can explain the case that some keywords are used in levels where this is not possible. As for example, the *for* keyword in level 1. We argue that this does not influence the results significantly and therefore the filtering is out-of-scope.

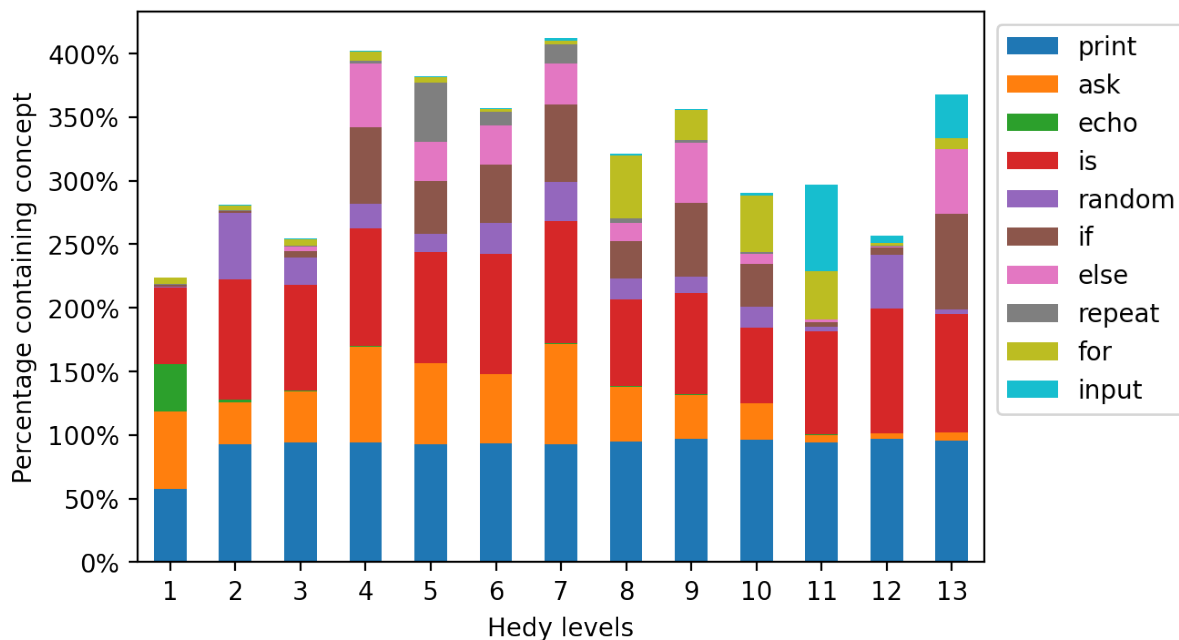

Figure 3: Normalized use of concepts over the first 13 Hedy levels

4.3 Language distribution

In this subsection, we look at the language distribution over the different levels. A figure similar to Figure 2 is created. However, a differentiation is made on the different possible languages. Currently, Hedy supports 10 different languages, for which the abbreviation can be found in the legend of the figure. The overview of the language distribution can be found in Figure 4. More languages have been added throughout the development of Hedy. Therefore, the visualization might not be representative for the current language distribution of use. In contrast to Figure 2 the data is filtered on unique sessions because we are interested in the user distribution and not the program distribution over the languages. However, it is clear that the languages Dutch and English are the most used languages. Due to the lack of usage of some languages, the Figure can be difficult to read. Therefore, an addition table is added given the exact values and percentages for each language, this information can be found in Table 7.

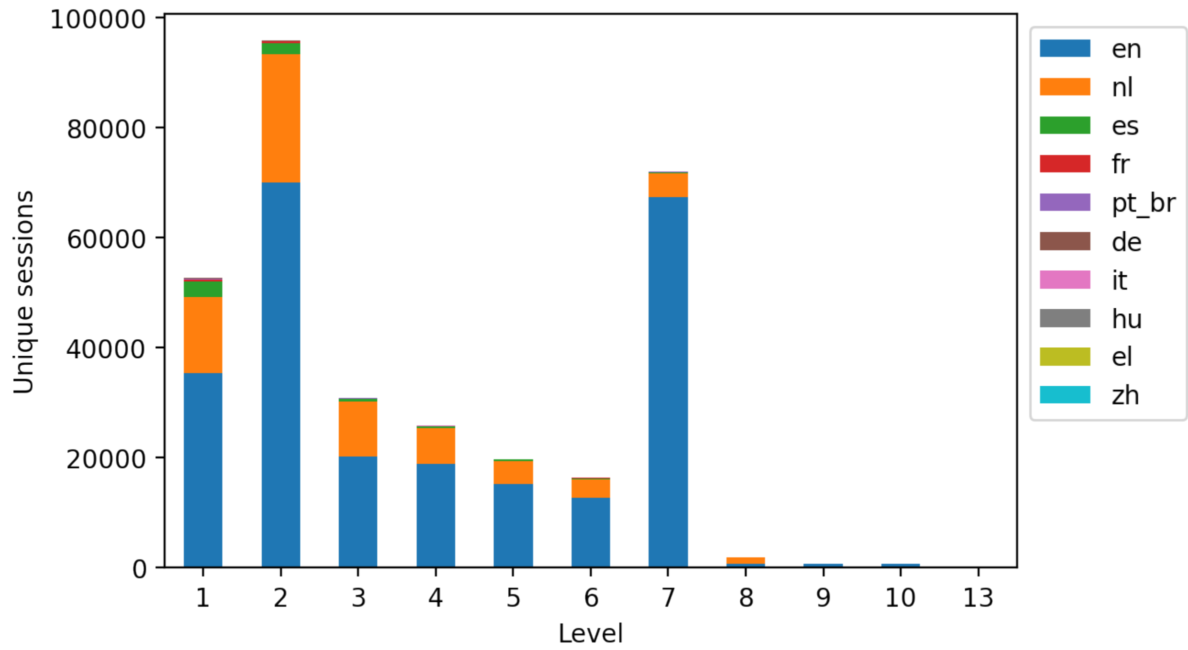

Figure 4: Language session distribution over the first 13 levels of Hedy

Language	Abbreviation	Programs created	Percentage
English	en	967822	80.04%
Dutch	nl	145791	12.06%
-	-	79440	6.57%
Spanish	es	11057	0.91%
French	fr	2317	0.19%
Portuguese	pt_br	1105	0.09%
German	de	807	0.07%
Chinese	zh	288	0.02%
Italian	it	187	0.02%
Hungarian	hu	0	-
Greek	el	0	-

Table 7: Hedy programs created per language

4.4 Program length and distribution

In this subsection, we analyse different length distributions throughout the levels. We take a closer look at three different types of distributions, namely the *character count*, *word count* and *line count* distribution. Each one is visualised using a box plot using SeaBorn and all three plots can be found in Figures 5, 6 and 7. Outliers are filtered for readability of the visualisation. Calculating the *character count* is done by using the Python built-in *len()* function of each submitted code, this means that the shown length includes white spaces as well. The *word count* is calculated by counting the numbers of white spaces in each submitted program *+1*. Similar, the *line count* is calculating by counting the amount of newline-characters (\n) *+1*. A general overview over all programs can be found in Table 8. Notice that this also included invalid programs, because a Hedy with one character or one word is not possible. For all three distribution a high standard deviation is found, combined with the high *Max* value for each distribution shows that larger programs are created then expected.

While we expect a positive linear correlation between the levels and the three different distributions due to the increasing syntax and programming functionality, it was expected that users would create larger and more complex programming along with the level. However, this correlation is not found. A Pearson correlation is calculated using the built-in *corr()* function of the Pandas library. A correlation of respectively 0.03, 0.02 and 0.01 is found between the Hedy level and the different length calculations, which can be classified as no correlation at all. The expected behaviour of programmers creating longer and more complex programs on higher levels is not supported by our analysis. We assume that this is due to the lack of programs created on the higher levels, whereas most in-class Hedy teaching focuses on levels 1 till 7. Using these levels as well to create larger programs.

Distribution	Min	Max	Mean	Standard Deviation
Character count	1	682,609	138.05	1286.51
Word count	1	680,679	28.08	799.13
Line count	1	16,080	5.49	59.79

Table 8: Hedy programs character, word and line count distribution

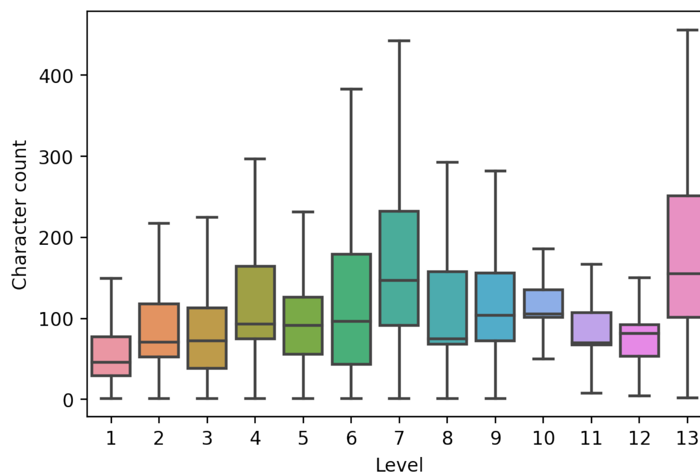

Figure 5: Overall character count distribution per Hedy level

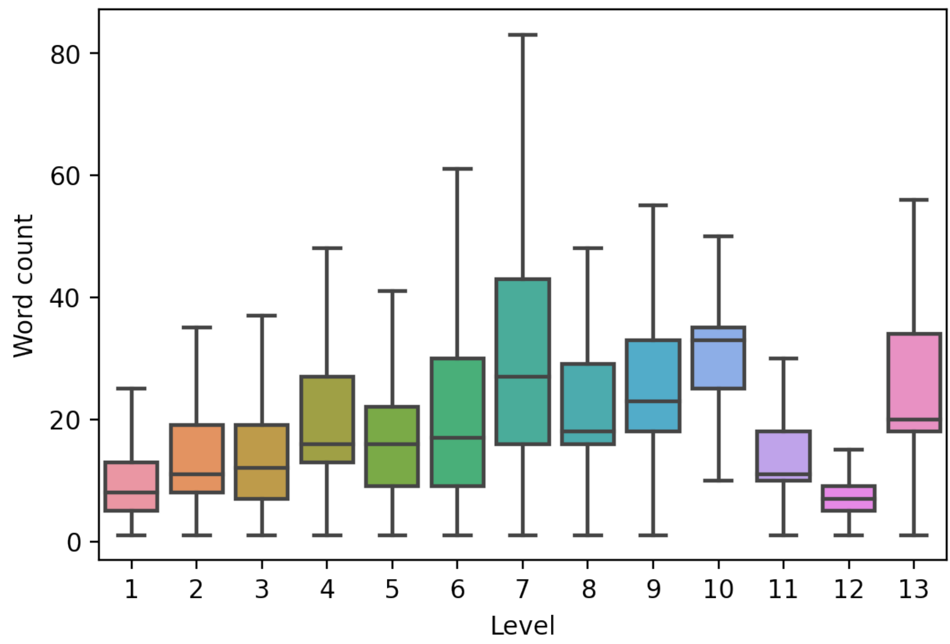

Figure 6: Overall word count distribution per Hedy level

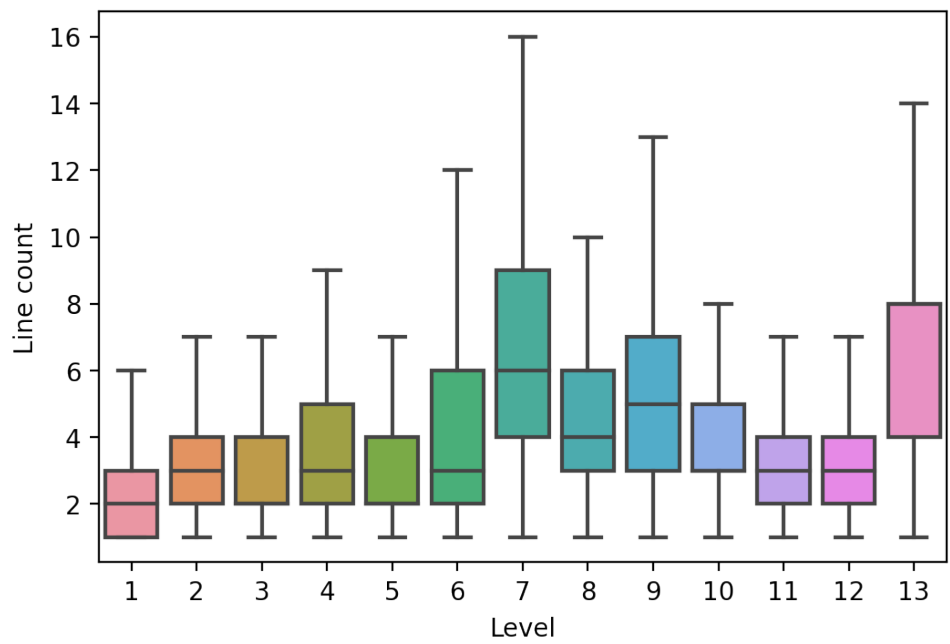

Figure 7: Overall line count distribution per Hedy level

4.5 Error analysis

In this subsection, we perform an *error analysis* on the Hedy dataset. First, we look at the general statistics on errors made, followed by a few examples of the most common errors. Then we look at the error distribution over the first 13 level of Hedy, and finally we give an overview of the error rates per language. By filtering on the "*None*" and "-" server errors, we keep a total of $293,977$ logs containing an error. Meaning that 24.313% of all the programs $(1,209,123)$ results in a server error. Similarly, we filter for the client errors: A total of $73,322$ client errors have occurred, representing 6.064% of the programs. We only look at the server errors because they are most likely caused by a programming error, while client errors can have very diverse causes. This happens when the parsing of Hedy code into Python code is successful and is returned to the users, but for some reason an error occurs on the client-side. This can either be a mistake in the parser (and not of the user) or another unknown connection mistake. We find a total of $37,738$ unique errors, for which we filter out the errors that only occurred once. Leaving a total of $10,269$ unique errors. Filtering is done because we are interested in frequent error patterns and not unique mistakes. The scale of the dataset makes it impossible to manually analyse and classify unique errors. Notice that due to the language-dependent error messages the actual number of unique errors is lower, however we are unable to compare these. An overview of some of the most frequent occurring errors is found in Table 9. By manually filtering all translations, the *Hedy errors* are retrieved, and the errors are counted, an overview for which can be found in Table 10.

Error	**Frequency**
String without quotation marks	11,329
Code at wrong level (level 2 at level 1)	7286
print is not a Hedy level 3 command. Did you mean print?	3497
Code at wrong level (level 7 at level 6)	3231
no code found, please send code.	2335

Table 9: Frequent errors found within Hedy code programs

Error name	Amount	Percentage
Parsing error	124,685	40.64%
Invalid command	83,542	27.23%
Python error	70,474	22.97%
Wrong level	12,793	4.17%
Unquoted text	12,299	4.01%
No code	1,906	0.62%
Incomplete code	718	0.23%
Unknown error	370	0.12%
Invalid space	2	0.00%
Undefined variable	0	-

Table 10: Overview of the error distribution on server errors in Hedy

One interesting frequent error is the print error occurring in level 3. With a total of 3497 occurrences, this makes up more than $1,2\%$ of all errors. Looking into this mistake, the error

is easy to explain: From level 3 and on quotation marks should be used when printing text, in contrast with levels 1 and 2 where text could be printed without any defining symbols. However, the *ask* keyword is still used *without* any quotation marks. As well as a variable; which is also printed without quotation marks.

When looking at the error percentage per level, all percentages are between the 15% and 60%. With a clear high outlier for level 11. One possible explanation for this high percentage could be due to an implementation bug at the moment of deployment through which correct code was faulty parsed and returned an error. One explanation for the low percentage of level 2 can be found in the introduction of the *random* keyword. When users have a correct code execution, they are probably interested in executing it multiple times for different results. This way the level error percentage will drop. An overview of the error percentage per level can be found in Figure 8. An overview of the error percentage per language can be found in Table 11. We argue that the difference in error percentage between English and Dutch is due to the amount of novice programmers using Dutch did so in a classroom environment, having for support for helping and solving errors. We are unable to find a cause for the (very) low error percentages of the less popular languages. The supported Hedy languages without programs in the dataset are left out of the table because this wouldn't add any information.

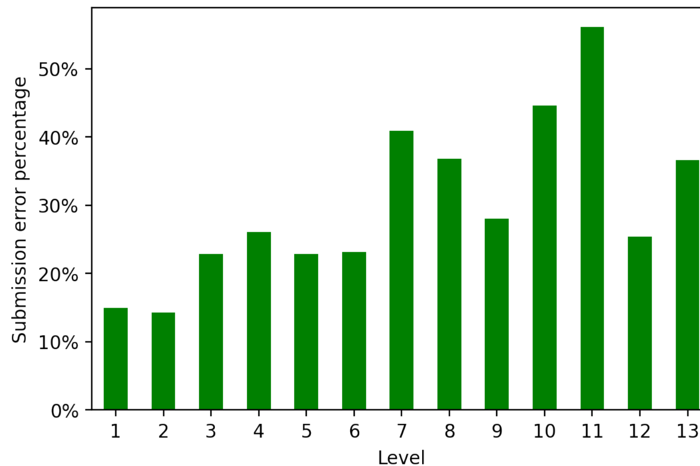

Figure 8: Overall error percentage per level

Language	Abbreviation	Code submissions	Server errors	Error percentage
English	en	967822	265324	27.41%
Dutch	nl	145791	25511	17.50%
-	-	79440	1408	1.77%
Spanish	es	11057	1133	10.25%
French	fr	2317	137	5.91%
Portuguese	pt_br	1105	123	11.13%
German	de	807	79	9.79%
Chinese	zh	288	13	4.51%
Italian	it	187	-	0.00%

Table 11: Error submission and percentage per language

4.6 Drop-out rate

In this subsection, we look at the drop-out rate of the users. We state that a programmer is counted as a drop-out when the last submission is one that results in an error, and no further attempts are made within the current or any other level within the same session. Simpler stated: When the last submission of a session results in an error, we count this as a *drop-out*. We assume that a higher drop-out rate suggest a lower usefulness of compiler feedback: giving the user not enough information to solve the problem and making them *rage-quit* their faulty program at hand. This rate will be seen as an important indicator of error message usefulness throughout this research.

A few side notes should be made on the drop-out analysis. Because the Hedy language itself is still in development, it might be the case that a faulty submission of code is done in the highest level available at that point. Suggesting that there is nothing wrong with the learning process, and they just stopped in the end with a faulty program. We leave this aspect out-of-scope because we state that it is difficult to filter this data and that it will have no significant influence on the statistics. It is still a drop-out, just a different kind. We could say a *non-learner* drop-out. Another important side-note is mistakes on the development side, because the language is still in development it is possible that in some versions correct Hedy code has been compiled incorrectly, resulting in an error. Something that would explain a drop-out because the code can't be fixed. We have only found this to be the case with a very small numbers of programs in higher levels, therefore we'll leave this out of scope as well.

We look at the drop-out percentage per level. The percentage is given as the percentage of the last submission of a session resulting in an error. A figure showing the drop-out rate per level can be seen in Figure 9. We see an almost linear increase of drop-out together with the higher levels. We argue that this can be explained by two factors: First, for most in-classroom Hedy sessions only the first levels are used, giving the novice programmers more help and making them less likely to quit. Secondly, the syntax gets more complex with the levels, making it harder to create successful programs on higher levels. We assume that this added complexity is an indicator for the higher drop-out rates at higher levels.

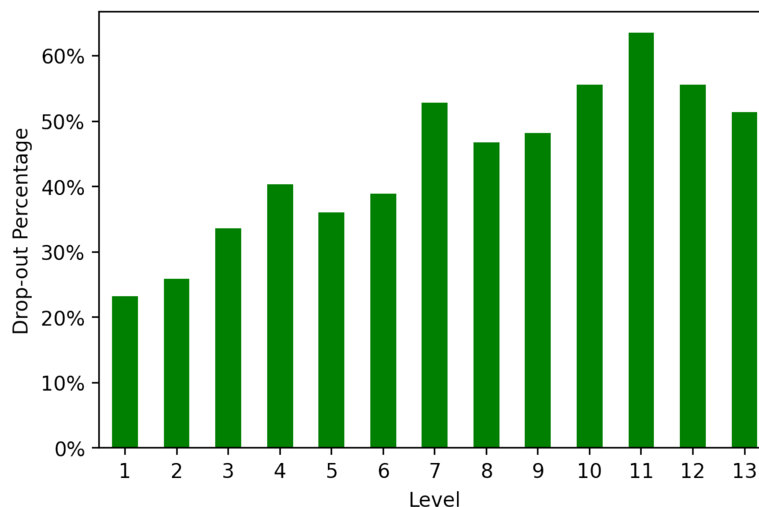

Figure 9: Drop-Out percentage per level

4.7 Identical submission analysis

In this section, an analysis on the identical (faulty) code submissions is performed. We state that the submission of identical code that results in an error is evidence of a misunderstanding of the error message. Making that, the novice programmer is unable to identify the mistake and solve the error by themselves. While we expect them to understand the concept of the computer simply following instructions, still another identical attempt is made. Even when it should be clear that the computer will respond identical and the attempt will result in another error.

We filter the data on containing server errors and group them on session, level and code. Meaning that all sessions submitting the identical code in the same *level* of Hedy will be in the same group. Then an analysis is done of the size of each of these groups: directly corresponding with the amount of times the identical code is submitted. For example, if a group has size 5 we know that that specific session on that specific level submitted that specific (faulty) code 5 times. The distribution of these group sizes can be found in Figure 10. It should be noted that no in-depth analysis is done on *why* the code is run several times. This might also be due to a bug in *Hedy* which returned an error on correct code. We argue that this part would not make an important difference in the results and is therefore out of scope. The most occurring amount of identical submissions is 2, occurring 20.528 times. The most times of re-submitting identical code is 526 times. As seen in the figure the distribution has an expected pattern of decreasing frequency over the amount of submissions. No surprising outliers are found, however the amount of 526 is larger than expected.

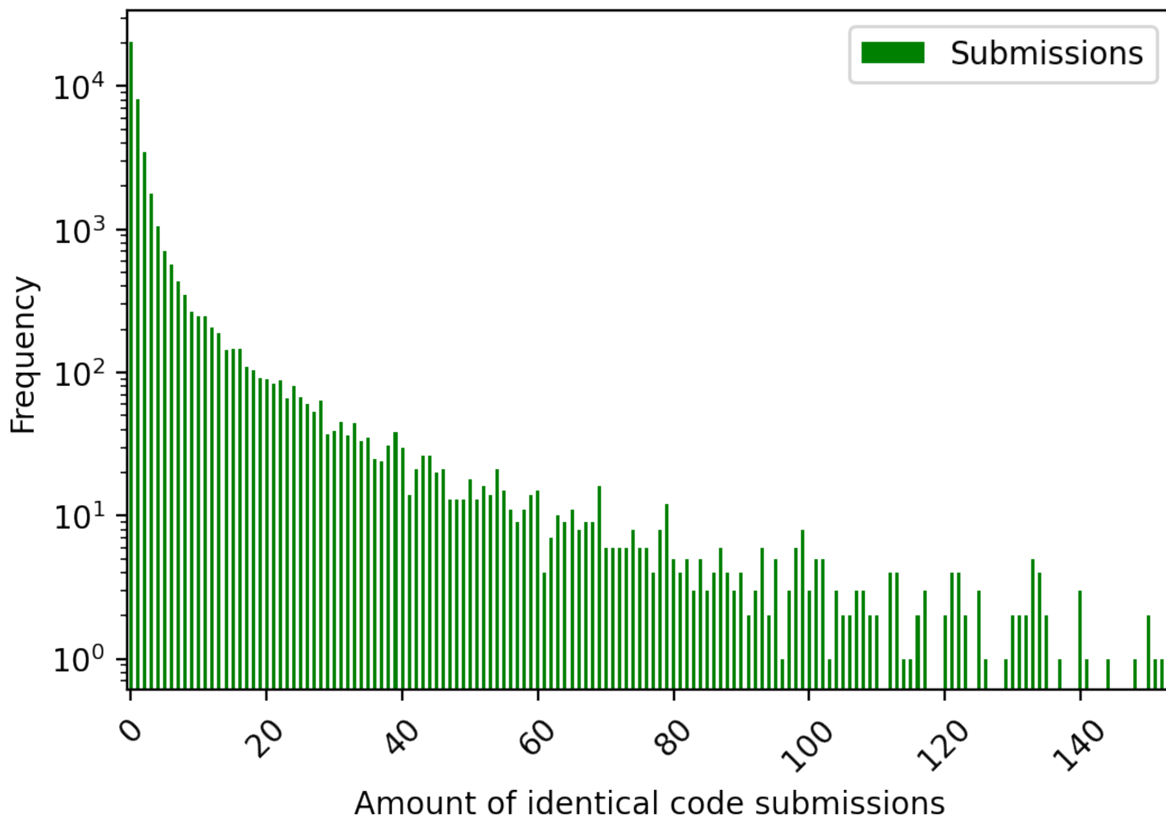

Figure 10: Overview of duplicate submission of identical (faulty) code

4.8 Time to Change

In this subsection, we analyse the time between two consecutive faulty submissions, so to get more insight if the programmers actually read the error message given. We differentiate between three time windows: A re-submission within 3 seconds is classified as *not reading* the error message, within 10 seconds is scanning the error message and between 10 and 30 seconds is actually reading and trying to understand the message. If the time-span is any longer, we think that the programmer takes a break, asks for help or in any other way does not directly continue with programming. Notice that in all cases there is some misconception in the perspective of the student, because *wrong* code is submitted again. However, through these different time-spans we can differentiate between expected scenarios such as *just pressing run again* or *reading but not understanding* the message. The time between submissions is visualized using a bar plot with bins of 1 second. Which can be found in Figure 11 and Figure 12. The visualisation seem to approach an expected distribution, except for the large amount of re-submission within 1 second. We argue that this is due to submitting the same faulty code and indeed, by filtering on identical code submissions, we find the distribution as in Figure 12. The found values and percentage for the three different time windows determined can be found in Table 12.

We find an unexpected large amount of consecutive mistakes being made after 30 seconds. Meaning that an error occurred, the novice programmer did some alterations and still another error occurred. This while taking quite some time to making the alterations. It is unclear why this percentage is high, and further research should be conducted in this *long-time* mistake making. For now, we are mostly interested in the 17.04% for which we deduct that they didn't read the message and made yet another faulty code submission.

Time till next submission	Consecutive mistakes	Percentage
Within 3 seconds	31,360	17.04%
Between 3 and 10 seconds	19,422	10.55%
Between 10 and 30 seconds	51,711	28.09%
Above 30 seconds	81,596	44.32%

Table 12: Consecutive faulty submissions per time window

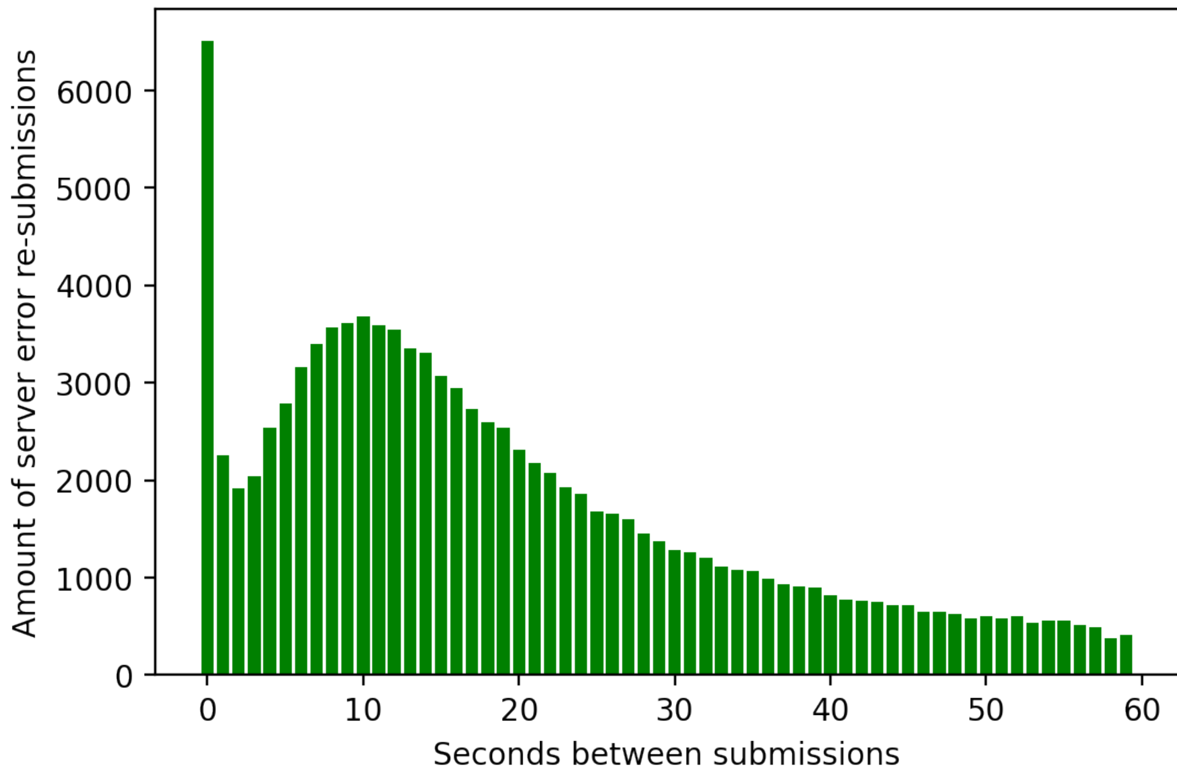

Figure 11: Overview of time between consecutive faulty code submissions

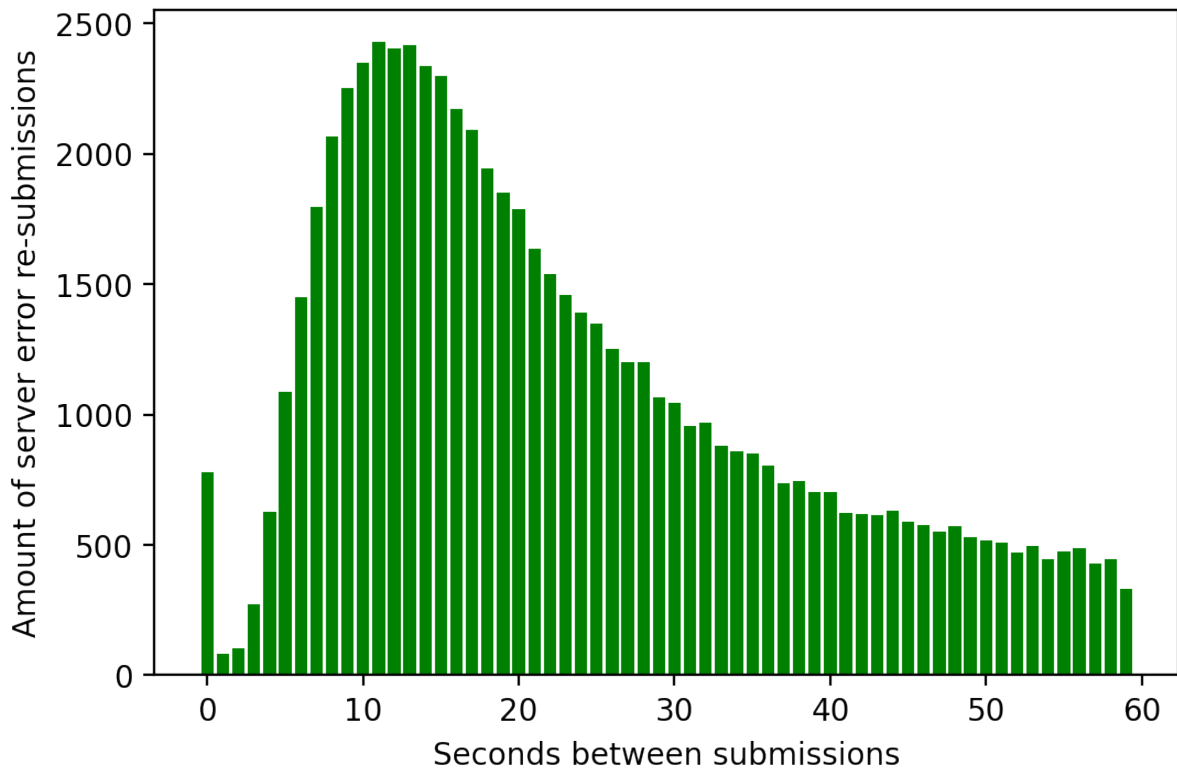

Figure 12: Overview of time between unique consecutive faulty code submissions

4.9 Steps to Success

In this subsection, we analyse the distribution of attempts needed to get a working program. We determine the *steps for success* as the amount of faulty code submissions by the same session before running a correct program. This statistic gives us insights in the usefulness of the error messages. Because we can argue that with good error messages, the novice programmer is able to determine the mistake and fix it within one attempt. A visual overview of the attempts needed for a success program can be found in Figure 13. The exact value of the first 5 attempts and the rest can be found in Table 13. The most attempts needed for a correct program were 516 attempts. It is interesting to see that while the overall error percentage is 24.313% still 93.66% of the first attempt, programs is correct. Suggesting that a user who made an error is far more likely to make another error than a new user. Giving us the insights that when making consecutive mistakes, it is harder to solve the problem at hand.

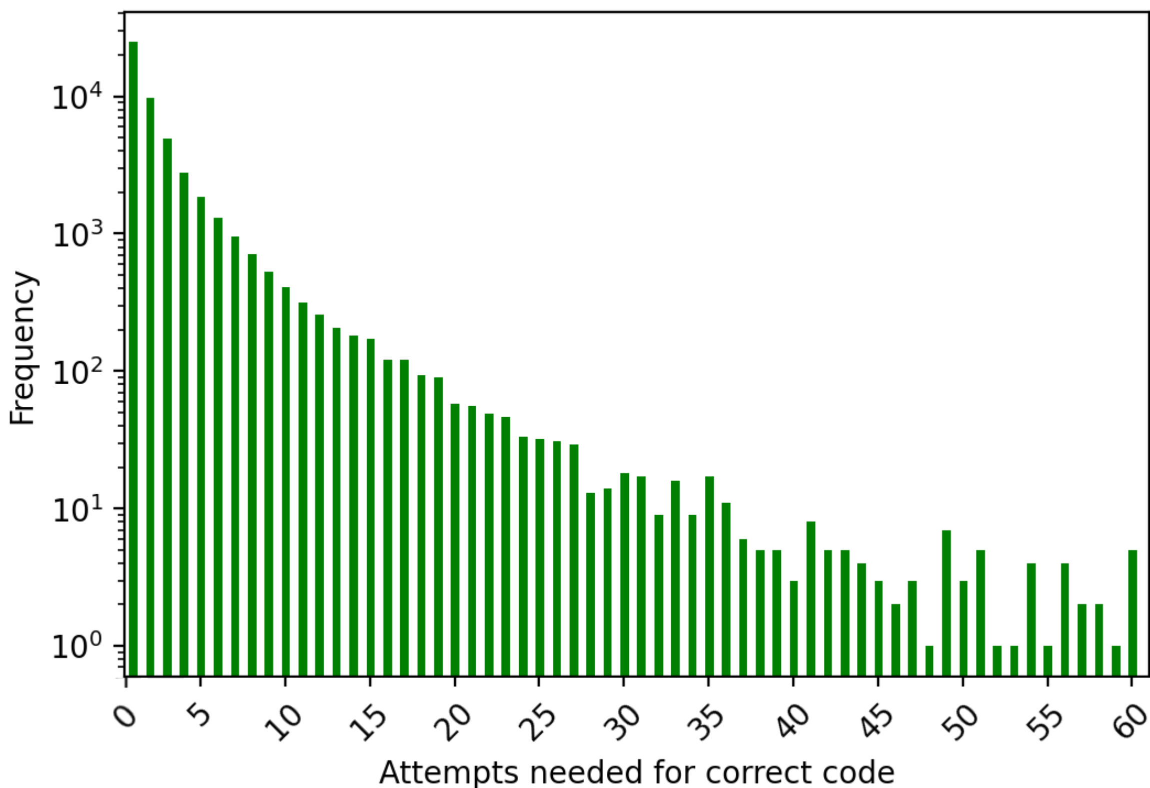

Figure 13: Overview of attempts needed for correct program

Attempts	Working programs	Percentage
1	741379	93.66%
2	24928	3.15%
3	9656	1.22%
4	4962	0.63%
5	2796	0.35%
More than 5	7870	0.99%

Table 13: Attempts needed to get a working program

5 Enhancing the Error Message

In this section we propose a solution to the found difficulties of novice programmers to learn to code due to the misunderstanding of the error message: The *Gradual Feedback Model* (GFM). A model that, similar to Hedy, works by introducing more explanatory errors dependent on the behaviour of the user. First, an introduction is given on the added value of this model. Then a literature review is conducted on similar work in the field of Enhanced Compiler Error Messages (ECEM). Follow by an explanation of how the model works and an explanation of the implementation within the Hedy web environment. It should be noted in this research the aim of the model implementation is on Hedy. While the model itself is language-independent it can be implemented in any programming language.

Imagine being a novice programmer, just starting your first programming course in Python. First exercise: output the sentence *"hello world!"*. You can imagine forgetting the quotation marks in the print command. Trying to execute the (faulty) code: *print(hello world)*. This is wrong, but what error message will you receive? The following error will be returned: *Syntax-Error: invalid syntax*. This makes sense, because a fault is made in the syntax. But what if we make a slightly different mistake? When trying to output the string *"hello"* also forgetting the quotation marks: *print(hello)* you will receive an *NameError: name 'hello' is not defined*. For an experienced programmer, we can spot the difference: the first is indeed an invalid syntax, while for the second example, Python expects a variable named *hello*. But: the novice programmer has no idea! In their mind, the mistake is identical, missing the parenthesis for printing a string. However, due to the large library of keywords, built-in functions and possibilities of modern programming languages improving this feedback to the user is difficult. Luckily, Hedy is a more simplistic language: enabling us to efficiently improve the error messages.

An earlier study on enhancing error messages to improve the debugging skills of novice programmers is done by Denny et al. Within an introductory programming class on Java, they implemented a modified *CodeWrite* tool which gave either the original compiler feedback or an enhanced version showing correct code as well as an explanation of the used functions or methods. According to the authors, *"Although we anticipated that the enhanced error messages would help students to identify and correct errors, analysis of the data shows no significant (or practical) effect."* [7] While these results don't sound very promising, they add a few side notes to their results, such that most mistakes might be easy enough to identify without enhanced error messages and where the compiler message already gives enough information for the student to solve the problem. Another side note is that they assume some students don't read the error messages at all.

While these results seem to suggest that enhanced error messages don't give the expected results, we assume that in the case of Hedy an improvement can still be made. First the interpreter feedback of Hedy is still in development and can be improvement informatively whereas the Java compiler has been in development for years. The note on *not reading the message at all* can be enforced by disabling the run button within the Hedy environment for a fixed amount of time. For which can be experimented with a *time* value that should be chosen carefully. On one side the reading should be enforced, where on the other side novice programmers shouldn't have the feeling that they have to wait on the interpreter while already understanding their mistakes.

5.1 Literature review

In this section, a literature review is done on related work in the field of improving error messages in programming languages. While some research might not be related to the structure of Python or Hedy, their mindset and solution to problems within different languages give a good overview of the work done in the field. The section is structured per-paper, for which each paper a short overview of the work is given. Finally, a short conclusion is given, as well as some perspective on the work that can still be done.

It is well-documented that novice programmers often struggle in understanding compiler error messages (CEMs) caused by incorrect syntax. [21] Three abbreviations often used in this research field of compiler messages are stated below. For simplicity and consistency, these abbreviations will be used in this research as well.

CEM = Compiler Error Messages
HCI = Human Computer Interaction
ECEM = Enhanced Compiler Error Messages

James Prather et al. analysed five different public experiments in the field of enhancing compiler messages, for which inconsistent conclusions were drawn. Interesting in this research is that it is performed from the perspective of novice programmers as well. They proposed five potential reasons for this inconsistency in research results. Which can be found in Table 14. Using a mixed-methods experiment in C++ they aim to address these reasons with an automated assessment tool, Athene. Their research did not show a substantial increase in student learning, but qualitative results indicate that students do indeed read the enhanced compiler error messages. [21]

Number	Reason
1	students do not read them
2	researchers are measuring the wrong thing
3	the effects are hard to measure
4	the messages are not properly designed
5	the messages are properly designed, but students do not understand them in context due to increased cognitive load

Table 14: Potential reasons for inconsistent conclusions on ECEM [21]

In 2012 Watson proposed using crowd-feedback to give users the feedback on their own level. Using the first found work of *gradual feedback*. First, the normal error message is given. When making the same mistake again, a more enhanced error messages is given. And finally, on the third mistake, a fix is proposed. They propose an online tool called *BlueFix*, which can be implemented within the BlueJ IDE: A well known IDE for novice Java programmers. They are the first research found that implements a framework with dynamic levels of support based on the compilation behaviour of the programmer. The flowchart of the interaction with their proposed tool can be found in Figure 14. The students perceived the tool as useful and an improvement in precision of 19.52% was found over other similar earlier research introducing the recommender system *HelpMeOut*. [8][26]

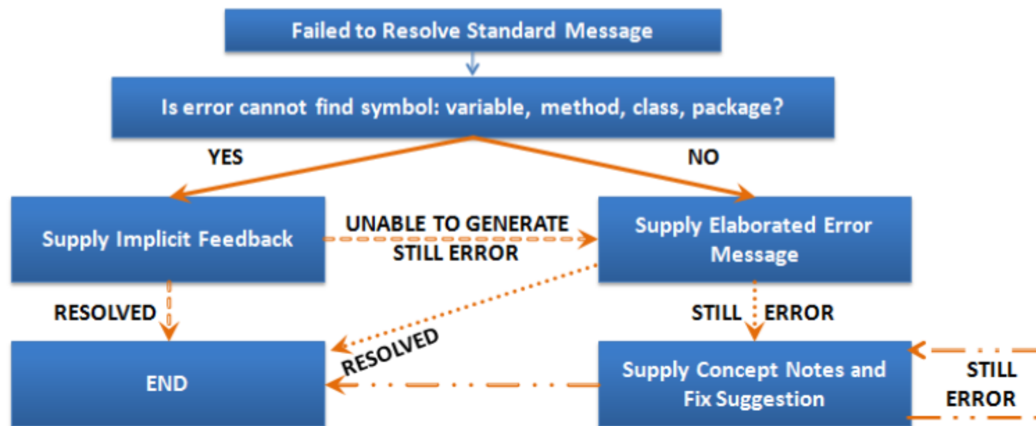

Figure 14: Flow chart of the BlueFix process, as proposed by [26].

Guillame Marcaeu et al. researched evaluation of error messages. Work that is useful for measuring the effectiveness of the usefulness of a specific error message. A *code rubric* is created with which they recommend changes to error messages: simplify vocabulary, be more explicit in pointing to the problem, help students match terms in the error message to parts of their code (e.g. using colour coded highlighting), design the programming course with error messages in mind (rather than an afterthought), and teach students how to read and understand error messages during class time. [18]

In the work by Traver et al. a more general comparison is made on messages gives by different compilers on the same language. While this does not apply for languages such as Python and Hedy due to having only one interpreter. It is interesting to read the approach on what compiler messages actually say and mean, combined with the question: Why are these messages so cryptic and a burden for programmers? They propose a set of eight principles that each compiler designer should keep in mind to make the messages useful, readable and understandable by the user. [25]

Raymon Pettit et al. performed research in the field on enhanced error messages as well. They implemented an enhanced compiler message for an C++ introductory course. They found no decrease in errors or any other difference in statistics with or without the enhanced messages. However, it should be noted that the enhanced messages were only given when code was submitted to the server to check the results. When compiling the code locally, the normal error messages were received by the user. Making it hard to drawn conclusive results on the usefulness of the enhanced messages because we can expect user to debug their code locally. [20]

Guillaume Marceau et al. use the DrRacket programming environment and by an in-depth analysis introduce a language independent rubric to evaluate student responses to error messages. They define the effectiveness of an error message by the following sentence: *Does the student make a reasonable edit, as judged by an experienced instructor, in response to the error message?* They start from a conceptual model where they formulate how an error message should help students. This in the sequence: Read $\rightarrow$ Understand $\rightarrow$ Formulate. They finalize a rubric which categories five ways of handling with an error, in each way how a step can be classified. [18]

Another interesting approach is matching error messages to stack overflow answers. Emillie Thiselton et al. take a look at this approach by implementing Stack Overflow answers as feedback for Python error messages. They implement a plugin for the Sublime IDE called *Pycee*. Two variants of the plugin are evaluated using a think-aloud study. Of course, this is an interesting approach due to the large availability of questions on Python. However, in the case of Hedy this is difficult due to two major parts: The significant smaller *syntax scope* and the small scale of use. [24]

Heemskerk researched the difference in the expected error information by novice programmers and the actual error messages provided by Python. He introduces the Error Message Component Framework (EMCF) to determine the structure of an error message. A total of 11 participants were asked to perform several Python tasks. He found that several error messages contained jargon or other terms that were unclear for the research participants. For example, the EOL (End-of-Line) error was only understood by one of the participants. One suggestion he makes to solve this issue is by creating a custom compiler / interpreter aimed at novice programmers. [9]

5.1.1 Conclusion

Some research has been conducted in the field of *Enhanced Compiler Error Messages* (ECEM). Varying from parsing messages to more human-readable text, suggesting fixed code examples and including *Stack Overflow* answers into the error messages. While all approaching shed an interesting light on the field of improving compiler feedback, rarely the user itself is the central point of interest. Almost all research (except [26]) look at the user as a general entity, making no differentiation between how programmers learn. While this might not be that big of an issue while trying to improve the feedback for experienced programmers, it is for novice ones. This shows that there is still a lot to research in the field of *Adaptive Feedback*, customized to the user. One possible implementation is suggested in the next section, the *Gradual Feedback Model* (GFM).

5.2 The Gradual Feedback Model (GFM)

Within a *normal* programming language, the goal of the interpreter or compiler is to point the programmer towards the mistake and explain why it was unable to execute the written code. While this is useful for an experienced programmer, compiler messages tend to be overwhelming and difficult to interpret by novice programmers. We now know that simply being pointed at a mistake is not a useful way of learning and understanding new things. With an added difficulty that it can be the case that the code is not programmed wrongly at the point of error, but a mistake at another point is made. Different research has been conducted on improving the compiler feedback, being more useful and explanatory towards novice programmers. However, this doesn't help with the difficulty for novice programmers: Pointing to mistakes doesn't help with the learning aspect of a (programming) language.

In this section, we propose a new way of giving error messages to novice programmers. Instead of just giving an error message, several *feedback levels* are proposed. Together with design changes to enforce error message reading and preventing duplicate faulty code submissions. It should be kept in mind that the aim of this model is the improvements of *learning* to program. The usefulness of the model is argued to be less and less the more experienced a programmer becomes. In this way it matches with the language of Hedy: Give gradually more syntax and options to prevent a high cognitive load and improve the learning aspect of programming.

5.2.1 Overview

The goal is to provide users with a *Feedback Model* that provides users with an error message corresponding to their level of help necessary, ensuring that the aspect of learning from mistakes is the primary goal of the given error messages. The model contains several levels of feedback, each giving another level of explanation on the mistake made. Connected to each level of feedback is a question of usefulness, asked after a successful code submission. An overview of the model can be found in Table 15. The feedback is given as an *addition* to the already existing error message. Because the goal is learning to program and read and understand errors. We still want the user to become familiar with the original messages without getting too dependent on the ECEMs of the model. The feedback is given in a collapsed window, giving the user a choice to interact with the model or only use the already shown original error message. A graphical representation of the model using a flowchart can be found in Figure 15.

Level	**Feedback**
1	Give *original* error
2	Give enhanced error message
3	Proposed similar correct code from dataset
4	Repeat the new concepts of current level
5	Suggest taking a break, come back with fresh view

Table 15: Gradual Feedback Model (GFM) levels of feedback

The feedback level starts at 0 will be raised by one after each error made. After a successful code run, the level is reset to 0. This results in the user being *stuck* at feedback level 5 until a successful code execution is made. This design choice is made to prevent users from using the GFM as a debugging strategy. An example of the gradual feedback for Hedy level 4 can be found in Table 16. Notice that the feedback itself is programming language-dependent, and the current example is based on Hedy. In the case of a non-gradual language, the feedback of level 4 will have to be altered due to the lack of *levels* and *new commands* at different points of learning. One could delete the feedback level all together, or implement a *keyword-recap* based on the user behaviour. However, this is out-of-scope of this research.

Feedback level	Example message
1	prnt is not a Hedy level 4 command. Did you mean print?
2	You used a command that doesn't exist. Take a closer look if you haven't made a mistake by accident or used a non-alphabetical letter at a weird spot.
3	print 'hello world!'
4	Remember, the "if" and "else" commands are new in level 4.
5	You seem to be stuck at this level, take a little break and try it again later.

Table 16: GFM feedback messages in Hedy level 4

When the user attempts to submit consecutive identical faulty code, an *Identical Code error* is returned and the feedback level is not raised. In this case, a short explanation is given on why the code will still return an error, identical to the previous attempt. An additional measure is proposed where the *run* button is greyed-out for a limited amount of time to enforce error-reading. A visual overview of the process flow of the GFM can be found in Figure 15. Notice that the *enforced error-reading* time is not defined and implementation dependent. An in-depth explanation on the Hedy implementation is given in Section 6.

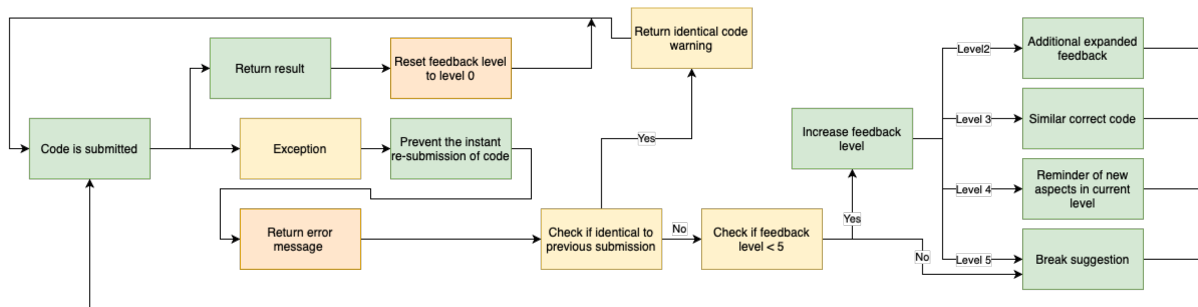

Figure 15: Language-independent GFM flow chart

5.2.2 Getting the feedback

In the case of a successful execution, directly after a feedback level higher than level 1 we will ask the user feedback questions dependent on their interaction with the model. This to enable us to evaluate the usefulness of the model. One general question is asked as well as a feedback level dependent question for each of the feedback levels the user has had interaction. Interaction is measured as expanding the feedback window (enabling the user to read the message). Through this feedback, we are able to evaluate the model. All questions are yes/no questions to lower the effort of the programmers to answer the question. We assume this will increase the participation willingness to answer the questions. An overview of the questions connected to each level can be found in Table 17.

Feedback level	Question
1	-
2	Did the extended explanation help you to solve the error?
3	Did the similar code help you to solve the error?
4	Did the recap on the new elements help you to solve the error?
5	Did the break suggestion help you to solve the error?

Table 17: GFM feedback level dependent questions

5.2.3 Design choices

In this subsection, the design choices of the model are discussed. By combining the knowledge gathered through the literature review, the errors found through the data analysis and our own ideas, the current model was developed. Firstly, the feedback given at feedback level 1, the goal is to guide the novice programmers towards Python. By giving the *normal* generated error first, they learn to recognize and work with short and sometimes cryptic errors. It might not be successful at first, but that skill will be learned by programming more often.

The feedback at level 2 will give more information on the error, but doesn't give too much away on how to exactly solve this error. It can be seen as a more *human approach* to the original error. Which is important, because a large aspect of programming is the skill of a programmer to *divide* a problem into smaller problems and solve these *systematically*. We want the novice programmer to better understand the mistake, without undermining the learning process of these two important aspects.

For level 3, we don't want novice programmers from becoming too dependent on similar correct code. When their programs become larger and the structure more complex, this is not a reliable way of debugging. By proposing the code only in level 3 and not earlier on, we assume that the novice programmers will see this as a *last resort* and not as a method of solving their programming problems. While it might be attractive to implement this feature as of level 1, we want to prevent this dependency. We explain this choice more in-depth in the following section.

Due to the structure of Hedy new concepts are introduced at each level, an aspect that increases the cognitive load. In level 4 of the model, we return to the programmer a reminder of the new concepts introduced in the current level. This reminder should help the programmer realise the difference between this level and the previous ones, enabling them to notice

mistakes faster.

Level 5 returns the advice to approach the problem differently and/or take a little break. While this doesn't bring them closer to the coding problem or give them a more useful error message, we still argue that this is important advice to return. It also prevents students from getting stuck into a repetitive pattern of failing to get a working program. It should be noted that the programmer still receives the original error message. So while this does not introduce new aspects or useful information, they are still able to use the original error message to solve the error at hand. The feedback level won't reset after make a mistake in this level of feedback to prevent using the GFM as a debugging strategy. The only way to reset the model is by either starting a new session or switching levels.

5.2.4 Feedback as a debugging strategy

In this subsection, the expected interaction of the programmers with the GFM are discussed. The model implementation is not transparent, it is unclear for programmers as to what moment they can expect what type of feedback. For the more observant ones, they might notice a pattern over time in which there is a structured order. This way they might be triggered to make mistakes on purpose and get their desired *level* of the model. One thing noticed by other, similar, research in the field of ECEM is that the messages are used as the debugging strategy itself. Making mistakes on purpose without a serious attempt at solving the problem to receive more information on the mistake then with the *normal* error message and use this to solve the problem. This is undesired behaviour: we want to provide the programmers with additional information to enable them to solve the problem themselves. Using this as a debugging strategy should be avoided. Due to the structure of the GFM implementation, this is difficult because the feedback level only resets at a successful code execution. We assume that this implementation choice will help the programmers in the long-term by not becoming dependent on the ECEMs provided by an additional model, something that isn't always there when they are programming. Resetting the feedback level by switching level, browser or in any other way enforce a new session is not expected behaviour and out-of-scope of this research.

6 Implementation

In this section, the implementation of the Gradual Feedback Model (GFM) implementation within the Hedy web environment is discussed. First, the current error handling is explained, followed by changes made to the Hedy web environment for the GFM implementation. Then we look at the implementation of finding similar correct code, the feedback that is given at feedback level 3. Followed by the front-end and logging implementation. Finally, we discuss the limitations & challenges of the implementation in Hedy. The GFM Hedy implementation is implemented in both English and Dutch and the forked-project of the Hedy source is open-source available on GitHub. [3]

6.1 Overview

Hedy is built upon Python and runs on *Flask* web server. The Flask architecture enables us to keep track of session variables, similar to how sessions work within PHP, these values can be accessed and altered throughout the whole website. This feature is the key part of the implementation of the GFM within the Hedy environment. Enabling us to keep track of the *feedback level* and several other values such as *previous code*. A value which is essential for preventing *duplicate faulty code submission*. At retrieving the home page, the *feedback level* value is set to 0 and increased at each consecutive error. The *html*-templates are altered to enable the addition of a *feedback box* containing the GFM messages, as well as enabling us to present the user with feedback questions on the model performance. An additional logging function is written to store the user interaction with the model.

6.2 Error handling

In this subsection, we discuss the error handling of Hedy. First, we discuss the current way Hedy handles errors with transpiling and executing code. Looking closer at the possible *exceptions* and how they are handled. Then we look at how the GFM is implemented within the Hedy environment. Expanding the error handling to keep track of additional values and return altered messages dependent on the HCI.

6.2.1 Current Hedy error handling

The largest part of the Hedy parsing and error handling works with a *try-except* statement. A *try* is used to *transpile* the code and successfully run it, then the result is added to the response which in turn is a JSON package that is returned to the webpage-template. In the case the transpiling fails, two possible exceptions can occur: either a *HedyException* or a general *Exception*. In the case of a *HedyException*, the *error code* is sent to a template to receive the corresponding error message. Hedy currently differentiates between 7 types of Hedy errors. An overview of the different errors and a short explanation can be found in Table 18. In the case of an *Invalid Space* error, the mistake is automatically fixed, and a *warning* is returned instead of an error. In all other cases, the error message from the *error template* is retrieved and sent to the user. The error templates are language-dependent, enabling the error messages to be the language of choice of the user. When a general *Exception* occurs, an unexpected event has happened because either the parser is unable to create either correct Python code or recognize a mistake within the Hedy code and return a *HedyException*. In this case, the Python error

itself is return to the user. An overview structure of the current Hedy error handling can be found in Figure 16.

Error name	Short explanation
Wrong Level	Correct Hedy code, executed at wrong level
Incomplete	Code incomplete, something is missing
Invalid	Invalid command is used
Invalid Space	Space where this is not allowed
Parse	Invalid code, unable to parse
Unquoted Text	Forgot quotations when printing text
VarUndefined	Trying to print an undefined variable

Table 18: Different Hedy Error types

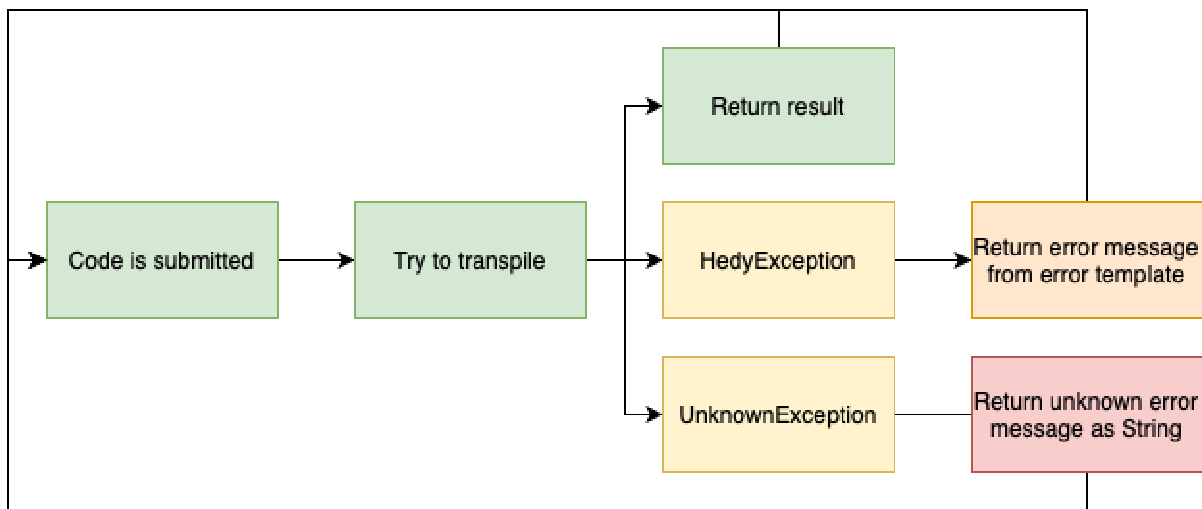

Figure 16: Current Hedy error handling implementation

6.2.2 GFM implementation Hedy error handling

For our GFM implementation, the except statement is altered. Whereas the error is still added to the response, several additional checks are performed. First, before execution, there is a check for identical faulty code, which will return an identical code warning and prevent the increase of the feedback level. Then there is a check for the current feedback level and the response corresponding to the level explained in Table 15 is added to the response. The action generated on that level is given as *additional* response. In the case of feedback level 2 the type of *HedyException* is retrieved and an *enhanced version* of the error messages is generated from the error messages template. A complete overview of these messages can be found in Appendix A: GFM Hedy Messages. In the case of feedback level 3 a Levensthein distance calculation is made on the code submitted by the user and a database of correct code, choosing the one with the lowest distance. In the case of a general *Exception*, the model works the same except for feedback level 2 of the model. Where an expanded *Unknown error* is returned from the error messages template. An overview structure of the GFM implementation for Hedy error handling can be found in Figure 17.

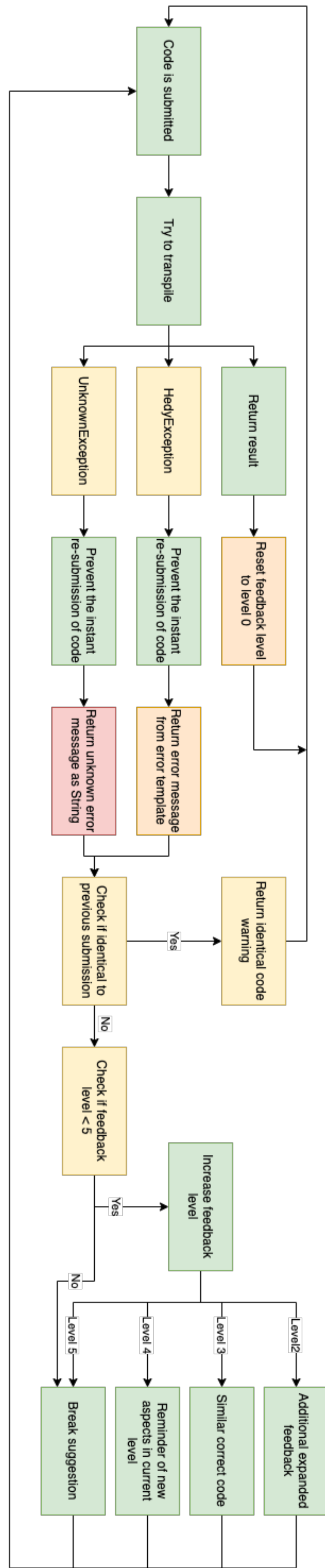

Figure 17: GFM implementation Hedy error handling

6.3 Finding similar (correct) code

In this subsection, we discuss the process of finding similar correct code. As explained in the model explanation in section 5, we present the user with similar code at *feedback level* 3 to enable a comparison and improvement strategy on their own (faulty) code. This similar code is retrieved from the original dataset on which the data analysis has been performed as well. First we focus on the process of pre-processing the dataset to be suitable for the similar code finding, then we take a look at the code execution done on run-time to find similar code.

6.3.1 Pre-processing the data

For each level and language we create a unique similar code file. Because the model is implemented in English and Dutch, we create two similar code files for each level. For example, *level3-en.csv*. We first filter on all code submissions not containing any error, this because we only want to present the end-user with correct code. Then we filter on all submissions made in our current language of choice, for example English. Followed by the creation of a concept-list for each level, containing all the keywords present in that *Hedy* level. Finally, a filter is added to filter out swear words and non-language words (either English or Dutch). This to filter undesirable language and personal information. For verifying if words exist in the current language, we use the *enchant* library. We split the current code submission on white-space and for each word iterate through a loop. If the current word is a keyword, we add it to the processed-code string. If and only if the current word is not a banned word and not a personal word, we add a *%* to our string. Otherwise, we break the current iteration and continue with a new one. Because for similarity we are not interested in the actual words but in the *keyword usage structure*, all non-keywords are replaced by *%*-symbols. Notice that any character can be used as placeholder, this is just as a unique character to filter on the similar code finding. If we successfully iterate through the code, the code and *processed code* are added to a list. After iterating through the whole dataset (of the current level and language) an addition duplicate check is done on the processed code, deleting all duplicate rows. An overview of the amount of rows deleted and remaining due to these steps can be found in Table 19 and 20. The files are only created for levels 1 till 10 due to a lack of data on the higher levels. The original code and pre-processed code are stored together on a line in the .csv-file. An example Hedy program can be found in Code Snippet 1 and the pre-processed in Code Snippet 2.

```
print Hello welcome to Hedy!
ask What is your favorite color?
echo So your favorite color is
```

Code 1: Pre-processing code example

```
print % % % % ask % % % % % echo % % % % %
```

Code 2: Pre-processing code result

Hedy level	Offensive words	non-English words	Duplicate Programs	Remaining Programs
1	1881	6113	10873	3731
2	2315	11442	26360	11792
3	955	5888	6213	3170
4	882	7077	1910	1852
5	947	5312	1819	1727
6	202	4792	1526	1437
7	618	28906	6126	4252
8	10	565	138	207
9	23	234	91	152
10	6	256	71	130

Table 19: General filtering statistics English files

Hedy level	Offensive words	non-Dutch words	Duplicate Programs	Remaining Programs
1	208	1725	2509	1006
2	64	772	2910	950
3	30	1172	1615	754
4	39	1469	281	401
5	23	1337	277	254
6	9	708	202	181
7	5	504	92	120
8	4	218	164	83
9	0	9	8	4
10	0	15	12	16

Table 20: General filtering statistics Dutch files

6.3.2 Pre-processing the submitted code

For pre-processing the (faulty) code on run-time, a similar approach is used to the one to pre-process the dataset. However, the swearing and non-language words filters are deleted because they are not relevant when the user creates a program. It is fine if users create a program containing personal information, made-up words or swear swords; we simply don't want to return programs containing them to users. The pre-processing step on the code itself is identical, we retrieve the level-dependent keywords and replace all non-keyword words by a *%* symbol. Then the Levensthein-distance is used to measure similarity against each row in the corresponding level and language .csv-file. A threshold of 75 is set on the implementation, if the costs of the algorithm are higher we return a *No similar code found* error. Similarly, for the case of levels 11, 12 and 13 for which no similar code file is create we return a *No similar code found* error. If the distance is 0 we directly return the current code because we are sure that no better suggestion will be found. The Python implementation of the on run-time algorithm can be found in Code Snippet 3. For finding the Levensthein-distance the *python-Levensthein* library is used, written in C. Due to the structure of the *Regex*-function, we first replace each non-keyword character in a word with a *%*-symbol and later on we replace a string of consecutive %-characters with only one %-character. This because the length of a non-keyword word is irrelevant for finding similarity on keyword-structure level.

```
concepts = get_concepts(int(level))
words = code.split()
code = ""
for word in words:
    if word not in concepts:
        code += re.sub(r"[a-z|A-Z|0-9|!?,''{}]", "%", word)
        code += " "
    else:
        code += word + " "
code = code.split()
temp = ""
for processed in code:
    if "%" in processed:
        temp += "% "
    else:
        temp += processed + " "
return temp
```

Code 3: Pre-processing code executed on run-time

6.3.3 Limitations

In this subsection, we discuss the limitations of the similar code finding implementation. One limitation of this similar code approach is the lack of performance on misspelled keywords. In the case of a mistake with misspelled keywords, they are not recognized by the model as *keywords* and therefore replaced by a *%* symbol. In this case, the proposed similar code can differ greatly for the faulty code of the user and does not help to solve the error. An example of a situation where the model does not have added value can be found below. We deemed this out-of-scope for this research, but it might be an interesting future work to improve the implementation of the similar code finding. The implementation could be improved by writing an algorithm to guess the most likely keyword when a non-logical coding structure is detected.

Faulty example program:

```
prnt Hello welcome to Hedy!
```

Pre-processed result for similarity finding:

```
% % % % %
```

Suggested similar code by algorithm:

```
ask What is your name?
```

Another limitation is the non-weighted approach of finding similarity. In the current implementation, each non-identical letter between the user code and the possible similar code has a cost of 1, as is usual with the Levensthein algorithm. However, we can argue that it is more preferable to weight specific letters or keywords to make them more important in the similarity process. For example, using the error message from the parser to detect the mistake and add more weight to finding (correct) code that has high similarity with that specific part of the program. It should be noted that a simplified way of adding weights to the algorithm is already implemented through the pre-processing step of replacing non-keywords with the *%* symbol. This possible improvement is out-of-scope and should be implemented and tested in future work.

Lastly, a limitation is the linear implementation due to which a mistake with the identical *keyword usage structure*, the same similar code is returned. This due to the use of *.csv*-files for which the reading of the files always starts at the start of the file. One possible solution to this problem is the implementation of a dictionary that keeps track of a *collection* of the lowest distance programs and, after iterating choices, one at random. Another possibility is reading the whole file into memory and start the algorithm at a random line. As all limitations, we deem this out-of-scope of this research, and this is something as well that can be improved by future work.

6.4 The front-end

In this subsection, the front-end implementation within Hedy is discussed. As discussed earlier in section 5.2 the response of the model is returned as an *additional* message. Resulting in a double error message: the original one and the one generated by our Gradual Feedback Model. To implement this visually, an additional *box* is implemented as an overlay on the code editor. This added light-blue box is shown as *additional information*, right below the original error box. The *feedback box*, as we will call it from now on, has an *expand* option: giving the user the option to look at the additional information or not. We keep track of the user interaction with the box on each feedback level. An example of the original error message implementation and one with the additional feedback box can be seen in Figure 18. Notice that in this figure, the *feedback box* is still minimized. An expanded example of the feedback box on the Hedy code editor can be found in Figure 19. The title of the *feedback box* is feedback level-dependent, for example when the similar correct code level is reached the title reads *"Similar correct code"*.

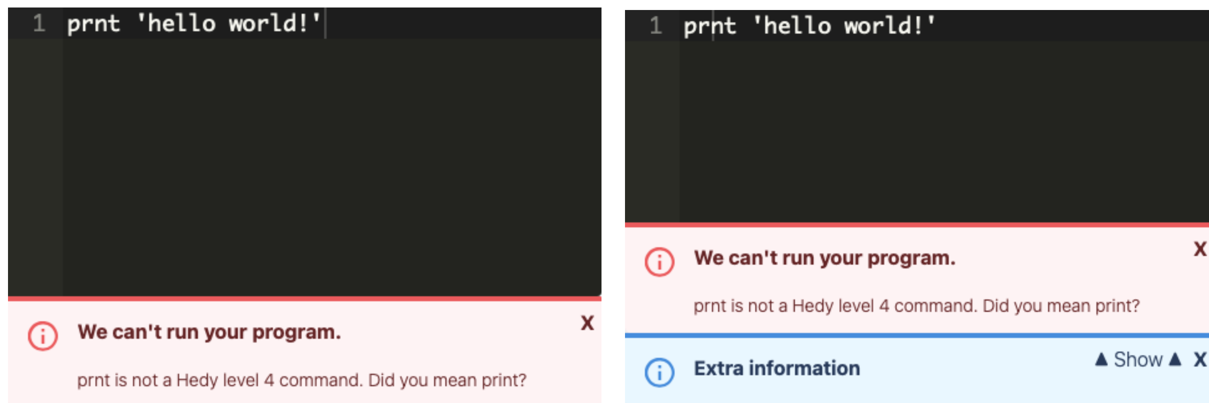

Figure 18: Comparison of the Hedy editor: left (original) and right (GFM implementation)

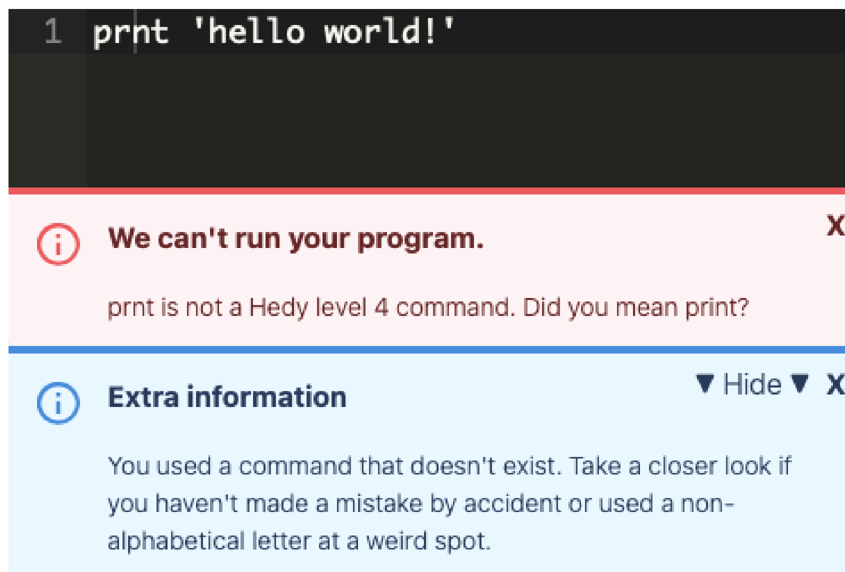

Figure 19: Hedy editor with GFM implementation (expanded)

6.4.1 Enforce error reading

To enforce the error reading, we decide to disable the *run*-button for a limit amount of time when an error is returned by the parser. As determined in the data analysis, we argue that re-running code within 3 seconds is classified as "Not reading the error message". Therefore, if a faulty code submission is made through the GFM implementation, an additional value is added to the JSON response storing the value for disabling the run-button. When the response is sent to the JavaScript for processing, a check is done for this value. If it exists, the run-button is disabled for 3 seconds and turned grey, while the *shadow* of the run-button is turned black. Gradually over the time-period the button returns to the original green colour whereas the shadow stays black until the last moment: then this is returned to the original dark-green colour as well, indicating that the user can execute code again. No additional information on disabling this button is given to the user, we argue that the visually changing of colours should be sufficient.

6.4.2 Retrieving user feedback

To determine the usefulness of the model by users, data analysis can only help us to a certain extent. Several assumptions will have to be made to state certain changes in behaviour as indications for being *useful*. To gather direct feedback from the users, an additional pop-up window is implemented with yes/no questions for the user. This pop-up will show if the GFM implementation has been shown to the user (more than one consecutive mistake is made) and the *feedback box* is expanded on one or more levels. When the feedback box has not been expanded, we assume that the user did not seem to think that the additional information could be useful. Indicating that they either saw the mistake themselves and fixed it, the interface was unclear, and they didn't notice the additional information, or they were *stubborn* and were sure they didn't need it. The questions will pop up after a successful code execution is performed, in order of user interaction. First, a general question is asked: *Did the additional information help you solve the error?*. Followed by level-dependent questions, which were explained earlier in Table 17. A question is asked for each feedback level the user had interaction with (and expanded the window). These answers are sent back to the server using an additional POST, after which a logging to the dataset is made to store the answer(s). In a case that the feedback box of a specific level is not expanded a logging is still made, the feedback related questions are then answered with *Null*. While this gives us less information, by logging this information we are still able to get more data on the problem-solving approach of the programmers. An example of the feedback questions pop-up window can be found in Figure 20.

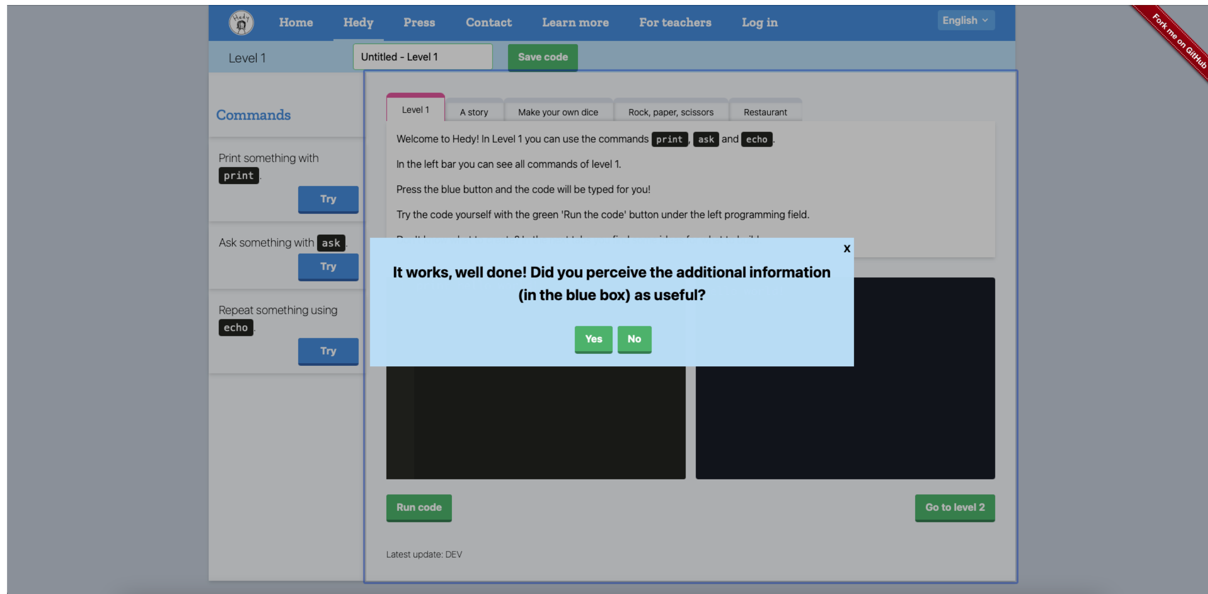

Figure 20: GFM general question within the Hedy environment

6.4.3 Preventing identical faulty code

Another aspect of the model is the prevention of executing identical faulty code. This is undesirable behaviour on the user side because they should be familiar with the concept that the computer always responds the same and submitting identical code is an indicating of not reading or not understanding the returned error. For the implementation, an additional value is stored with the *session*-values of the user. The lastly executed code is stored temporary if it resulted in an error. When a new program is submitted, an additional check is performed to verify if the new code differs from the last submitted code. Only if the feedback level is higher than 1, otherwise the previous submission didn't result in an error. This is checked before the actual parsing of the code. After the check, the code is still parsed to generate the corresponding error message from the Hedy parser. The original error message is still returned because we want this feature to be consistent with the GFM, returning the error message as well as additional information. Parsing isn't strictly necessary, because we code is identical we already know the error message from the previous submission and this could be re-used. However, this part of the implementation is deemed out-of-scope of this research. This could (and should) be improved in future work to prevent unnecessary server load. Additional to the error message, an *Identical Code* message is returned to the feedback box, explaining that computers will always respond the same to the same instructions. The exact message can be found in Appendix A: Table 29.

6.5 Expanding the log

To enable an analysis of the interaction and usefulness of the GFM implementation, several additional values will have to be logged. As one can verify by looking back at Table 4, Hedy currently logs 14 values after each code submission. Due to the structure Hedy is implemented, we are unable to merge the additional values in the already existing *logger* function. Currently, a new *log* is made after the parsing, but before sending the response to the front-end of the application. Because we want to store feedback and interaction of the user on the model, we have to wait for their response on the front-end. Therefore, an addition log is made after the front-end interaction, which can be merged onto the original log later on. The original logger function is slightly altered as well to enable the easier filtering of submissions for which the model is used.

Firstly, a *Boolean* value named *GFM* is added to the original logger function. This will be set to *True* if the model is used and *False* if the model isn't used. The later will be the case when a non-supported language is used while the model is active. In the case of A/B testing where the model isn't present (the original website is returned to the user) this additional value doesn't exist. The *feedback level* is also added as a value to the original function. This enables us to easier follow the error-solving path of the users. Similar to the GFM value, this will be empty when the original website is returned to the user. When consecutive mistakes are made, no additional logging performed. But, after the first successful execution within the same session and level, the earlier discussed feedback questions are asked to the user. These answers as well as information on interaction is logged using an additional function. All values stored through this additional logging process can be found in Table 21. The storing of session and date enables the manual merging later on for the data analysis.

Name	Data type	Example value
feedbackLevel	Integer	3
generalUsefulness	Boolean	True
feedbackUsefulness	Array of Booleans	[True, null, False, null]
similarCode	String	print Hello World!
feedbackCollapse	Array of Booleans	[True, False, True, null]
GFM	Boolean	True

Table 21: Additional GFM values stored through the additional logger function

GeneralUsefulness contains the user answer to the general usefulness question. *feedbackUsefulness* and *feedbackCollapse* both contain an Array storing the interaction with the user. The values in the array represent each level of feedback where the model produces additional information for the end-user, so level 2 till 5. *feedbackCollapse* stores if a user had interaction with the additional information. It is set to *True* if the user did expand the feedback box, *False* if the user didn't. If a successful code run is made before a level of feedback is shown, the value is set to *Null*. For the *feedbackUsefulness* variable, the user is asked a level-dependent question of each level of feedback with which the user had interaction. The answer is stored as either *True* or *False*. If the user didn't expand the feedback box of a specific level or the level was never reached, then *Null* is stored as well. If the user closes the question window without answering, an "-" is stored instead.

6.6 Limitations & Challenges

In this subsection, we discuss the limitations and challenges on the GFM implementation within the Hedy environment. One of the challenges was the lack of comments within the Hedy code. Making it difficult to understand and keep track of different aspects of the language. Getting to know the environment and understand how the dependencies worked were a large part of the implementation process. Due to the language and web environment still being in development complicated the process because error handling, error messages and dependencies kept changing over time. For a verifiable A/B test, it is essential for the GFM implementation to be visually identical to the most up-to-date version of the Hedy web environment to account for any external influence on the performance of the users. Therefore, the GFM implementation had to be updated several times to keep up-to-date with the public version of Hedy and be ready for deployment of the A/B testing. This was due to the ongoing development of Hedy itself: functionality and the user interface changed multiple times during this research.

Another limitation of the implementation within Hedy is the data handling. Due to the original logging being processed before the interaction with the user, we had no other choice than writing an additional logging function to store the required GFM data. Through this design choice, the additional logging is performed from the client-side to the server through an unencrypted JavaScript POST, making it vulnerable for the data being altered. However, we assume that this has not happened and have found no signals that this is the case. Another limitation of this logging process is that the additional logging is only performed after a *successful* code submission. So when a user coding in Hedy makes consecutive mistakes but quits before executing a successful program, the user interaction and feedback is not stored and/or asked. Making us lose valuable research data. This choice was made because otherwise it would be difficult to ask feedback questions on the interaction at each level of feedback. However, this could be improved in future work, so a logging is made after each user interaction. The user interaction should be stored server-side temporarily and be retrieved at each point of interaction to update the model-user interaction. While this would increase the server-load, the additional gathered data would be useful for analysis of the usefulness of the model.

Lastly, a limitation was the lack of programs created in the higher Hedy levels. Due to parallel research to this Thesis, additional levels were developed at the same time as the Gradual Feedback Model. The newest levels would contain unknown bugs, resulting in undesirable behaviour and faulty error messages, making it difficult to analyse the errors because we were unable to determine if the mistake was on the user or server side. Also, not many programs were created on levels 14 and higher, resulting in lack of data for creating similar code files and returning the users useful suggestions. Therefore, throughout development the choice was made to only focus on the first 13 Hedy levels due to these being documented well and no longer subject to change. Therefore, the similar code implementation should be improved and updated in future work by creating new similar code files on the newest dataset at hand, improving that level of feedback.

7 Results

In this section, the results of the *Gradual Feedback Model* (GFM) implementation within Hedy are discussed. As discussed earlier, the data is gathered using A/B testing, giving a percentage of users the original Hedy website while other users receive the Hedy version with GFM. This section is divided in two main analysis: a comparison analysis and a usefulness analysis. First, we compare general statistics on data gathered in both versions in the same period of time. It is important to analyse the same period of time because Hedy is still in development and other changes (coding errors, visual updates etc.) could influence our results. This analysis has a similar approach as the general data analysis performed in section 4, looking at error percentage, drop-out rate and code length deviation. The usefulness analysis will focus on the user feedback gathered through the model. As explained in section 6 for each code submission a value of the current *feedback level* is added to the log and on each successful submission an additional log is made containing all user interaction on the model. This data is analysed to get more insight in the usefulness on each level and the user interaction with the model. While some general statistics will also give insights into the usefulness of the model, this division into two parts is made to differentiate clearly on the data used for the analysis. For the comparison analysis the already existing values are mainly used, whereas of the usefulness analysis the focus lies on the additional values specifically store for the GFM implementation.

7.1 The data

The A/B test ran from June 15th 2021 till July 12th 2021, varying in percentage and duration. For example, the initial testing was performed with a 10% re-direct value. Meaning that 10% of the users are re-directed to the GFM implementation while 90% will still be shown the original Hedy website. For simplicity, we state that the A/B test ran from June 28th till July 12th at a re-direct percentage of 50%. The users were re-directed at random to one of the versions. A complete overview of the exact timestamps of A/B testing can be found in Table 22. Through this testing, a total of 4552 programs is created with the GFM implementation by 116 unique sessions. The model has been presented a total of 89 times to 48 unique sessions. Meaning that over 40% of the users has been presented with the model. Notice that this is not the same as *model interaction*, which will be discussed later on. For additional feedback to be counted as *interacted*, the user will have to have the feedback box expanded to enable the reading of the message. A visualization of the feedback level at the moment of code submission can be found in Figure 21. Showing that the largest part of the programs is submitted with feedback level 0. As the feedback level is updated *before* the logging process, we know that all these programs are correct programs. We see an expected pattern of decreasing code submissions per feedback level as it is expected that more errors are solved throughout the debugging process, re-setting the feedback level. The increase in feedback level 5 is expected because the feedback level is only reset at a correct submission or when switching Hedy levels. So, with more than 5 consecutive errors, the user will still be presented with the corresponding feedback of level 5 of the Gradual Feedback Model.

Start date	End date	Re-direct percentage
June 15 13:18	June 15 14:18	1%
June 17 11:34	June 17 11:36	10%
June 17 14:34	June 17 14:36	10%
June 18 15:34	June 18 15:39	50%
June 21 11:00	June 21 12:00	10%
June 21 13:00	June 21 14:00	10%
June 28 10:56	June 28 15:00	50%
June 30 12:34	July 1 11:20	50%
July 5 10:23	July 9 09:13	50%
July 11 22:48	July 12 19:45	50%

Table 22: Timestamp overview of GFM A/B testing within Hedy environment

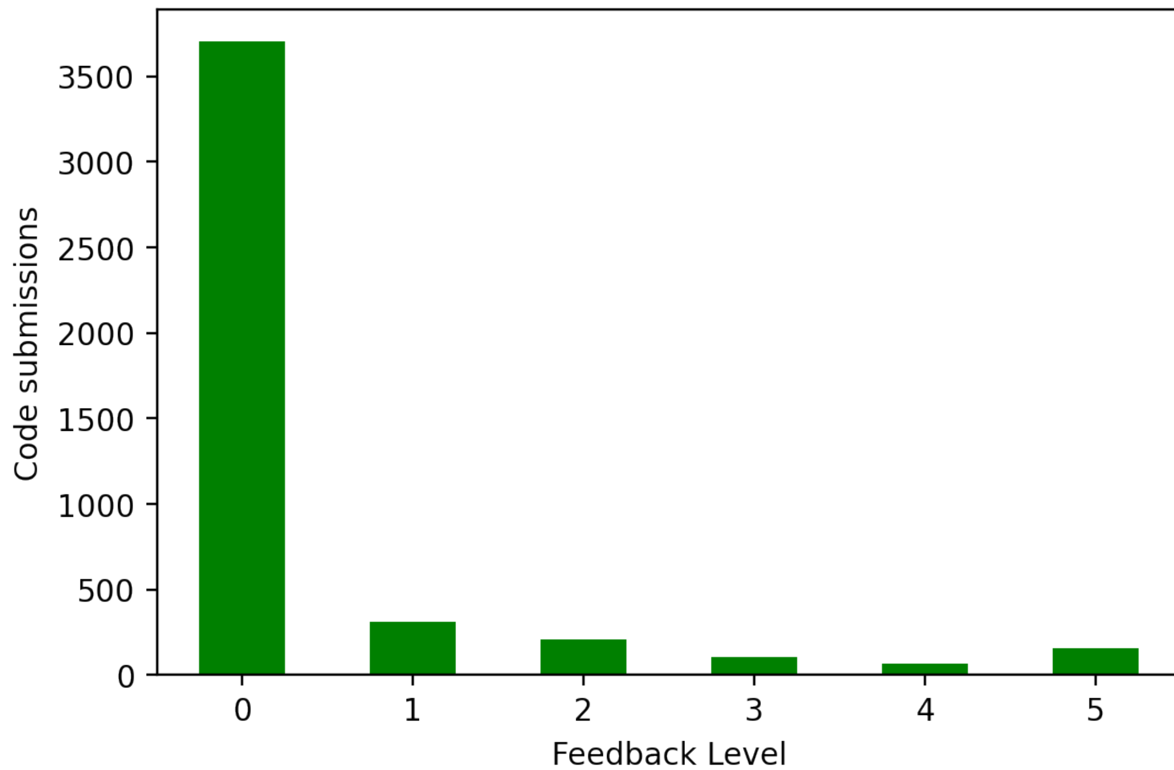

Figure 21: Code submission distribution per feedback level

7.2 Comparison Analysis

In this section, we compare general statistics on the data gathered through the original Hedy website and the one with the GFM implementation over the same period of time. For simplicity, we use all data gathered using the model and for the comparison use all data gathered through the original Hedy website from June 28th till July 12th. While this does not exactly correspond with the time-windows of the A/B testing, this shouldn't influence the results because no other updates have been implementation in any of the two versions of the period of the A/B testing.

7.2.1 Error rate

In this analysis, we look at the error rate at each level. The error rate is calculated as the normalized amount of programs that result in a server error on a specific level. As with all analysis throughout this research, we only look at the first 13 levels of Hedy. For the original Hedy website we find an overall error percentage of 20.03% whereas for the GFM implementation this is 16.89%., a slight improvement on the model. A visualisation of the error rate per level for both the original website and the GFM implementation can be found in Figure 22. Interestingly, the original implementation has a higher error percentage at the lower levels, whereas the GFM implementation has a higher error percentage at the higher levels. Notice that only the first 11 levels are shown because no (faulty) programs were created in either level 12 or 13 throughout the time of A/B testing.

We perform a statistical significance analysis to calculate if the decrease in error rate is indeed significant. The analysis is based on the significance test for comparing two proportions in categorical data from the textbook on Statistical Methods for the Social Sciences. [1] The sample size of the original implementation is 7618 programs with an error rate of 20.03% and the sample size of the GFM implementation is 4552 programs with an error rate of 16.89%. We assume a desired confidence level of 95% for which we calculate the Z-score of our samples using $Z = (p_1 - p_2)/\sqrt{[p(1-p)(1/n_1 + 1/n_2)]}$. Whereas n_1 and n_2 are the sample sizes and p_1 and p_2 the proportions of measurement (in our case the error rate). This results in a Z-score of 4.282. The corresponding Z-score to a confidence level of 95% is $Z_{\alpha/2} = 1.96$. Because $Z > Z_{\alpha/2}$ we can state that the decrease in error rate is indeed significant with a confidence of 95%.

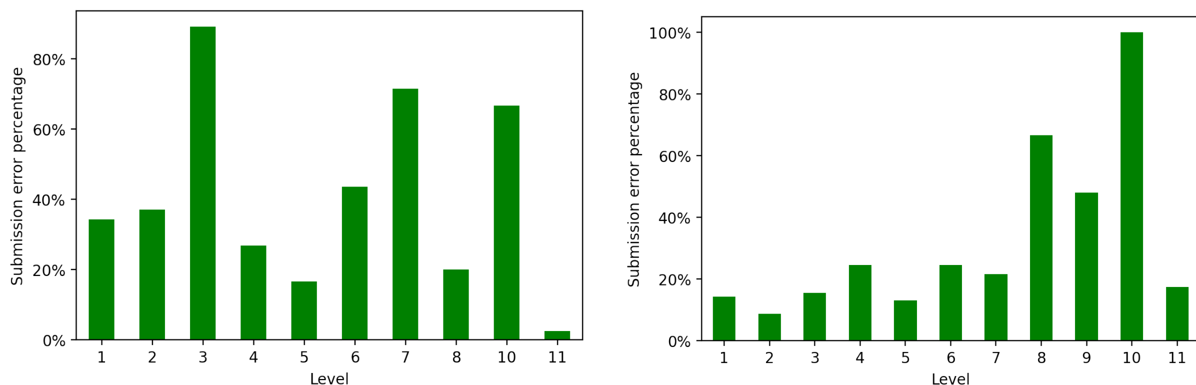

Figure 22: Comparison of error percentage: left (original) and right (GFM implementation)

7.2.2 Drop-out rate

In this analysis, we look at the drop-out rate at each level. A drop-out is defined as *True* is the last submission of code within a specific level run by a unique session results in a server error. In any other case, the drop-out is defined as *False*. This is identical as the drop-out analysis performed earlier in the Hedy data analysis. Similar to the error-rate analysis, a normalized value is calculated for each Hedy level. A visualisation of the drop-out rate per level for both the original website and the GFM implementation can be found in Figures 23 and 24. Notice that the levels in the graph without any bar show that no drop-outs are found according to our definition. The sample size of the original implementation is a total of 443 session/level combinations for which we find 81 having the last submission result in an error. This results in an overall drop-out percentage of 18.28%. For the GFM implementation, we find 198 session/level combinations for which we find 50 having the last submissions result in an error. This results in an overall drop-out percentage of 25.25%. The cause of this higher drop-out rate is unknown and requires further research in future work to better understand the motivation behind actions of users. Notice that the session/level combination results in more samples than the amount of sessions, one session is able to run code on multiple levels.

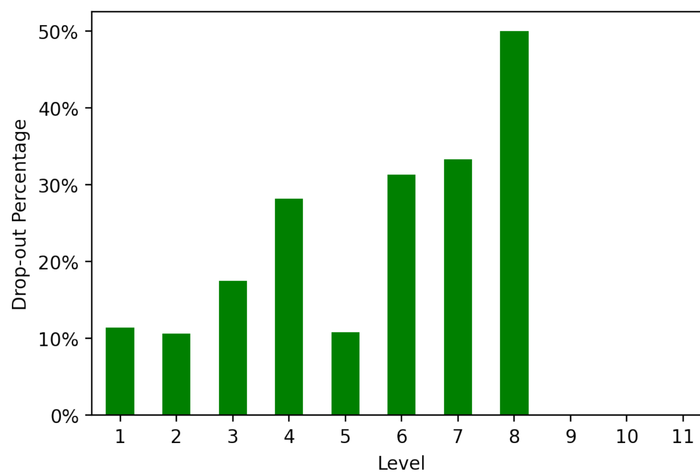

Figure 23: Drop-out percentage per level (original implementation)

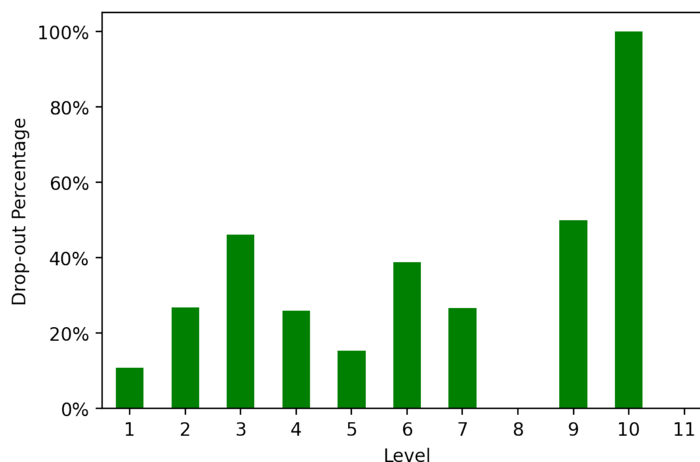

Figure 24: Drop-out percentage per level (GFM implementation)

7.2.3 Steps to success

In this analysis, we look at the amount of submissions needed to submit a correct program. This is calculated per session/level combination and visualized using a bar plot with bins per 1 step, similarly to the general data analysis performed on the original Hedy dataset. A visualisation of the frequency of steps before a successful submission for both the original website and the GFM implementation can be found in Figures 25 and 26. For the y-axis, a logarithmic scale is used due to the high frequency of 1 attempt needed. Visualising on a linear scale would make the figures difficult to read. The most attempts needed on the original implementation are 67 and 25 on the GFM implementation. However, due to both these values only occurring once, we can classify them as outliers. For both implementations, we look at the distribution of attempts needed with a focus on the first 5 attempts because the GFM implementation returns additional information for the first 5 consecutive mistakes, aiming to help the users at these points.

For the original implementation, we find a total of 6092 session/level combinations for which 5714 are successful on the first attempt, followed by 170 and 68 on the second and third attempt. For the GFM implementation a total of 3783 session/level combinations is found, for which 3574 are successful on the first attempt, followed by 93 and 37 on the second and third attempt. An overview of the *steps to success* comparison of both implementations can be found in Table 23. We perform a statistical significance analysis similar to the one performed on the error rate. However, we are now interested if the found difference in the total distribution on attempts is significant. For the error-rate, we only had categorical data: either an error or no error. In this case, we can also calculate the mean and standard deviation of the attempt's distribution: therefore, a T-test is used instead of a Z-test. The used formula is slightly different, namely: $T = (\overline{y_2} - \overline{y_1}) / \sqrt{(s_1^2/n_1) + (s_2^2/n_2))}$. Where $\overline{y_1}$ and $\overline{y_2}$ represent the mean (of attempts needed) for both implementations, and s_1^2 and s_2^2 represent the standard deviation. n_1 and n_2 are the sample sizes. Due to the large sample, $n > 30$ we can assume the t distribution to follow a standard distribution. [1] Calculations of the T-score we find 1.6965. As we assume a standard distribution, we know that a T-score of 1.96 is needed for a confidence of 95% on the statistical significance. As $T < 1.96$, we conclude that there is no statistical significance in the decrease of *steps to success*.

Attempt(s)	Frequency	Percentage
1	5714	93.80%
2	170	2.79%
3	68	1.12%
4	44	0.72%
5	22	0.36%
6 and on	74	1.21%

Attempt(s)	Frequency	Percentage
1	3574	94.48%
2	93	2.46%
3	37	0.98%
4	26	0.69%
5	14	0.37%
6 and on	39	1.03%

Table 23: Steps to success: left (original implementation) and right (GFM implementation)

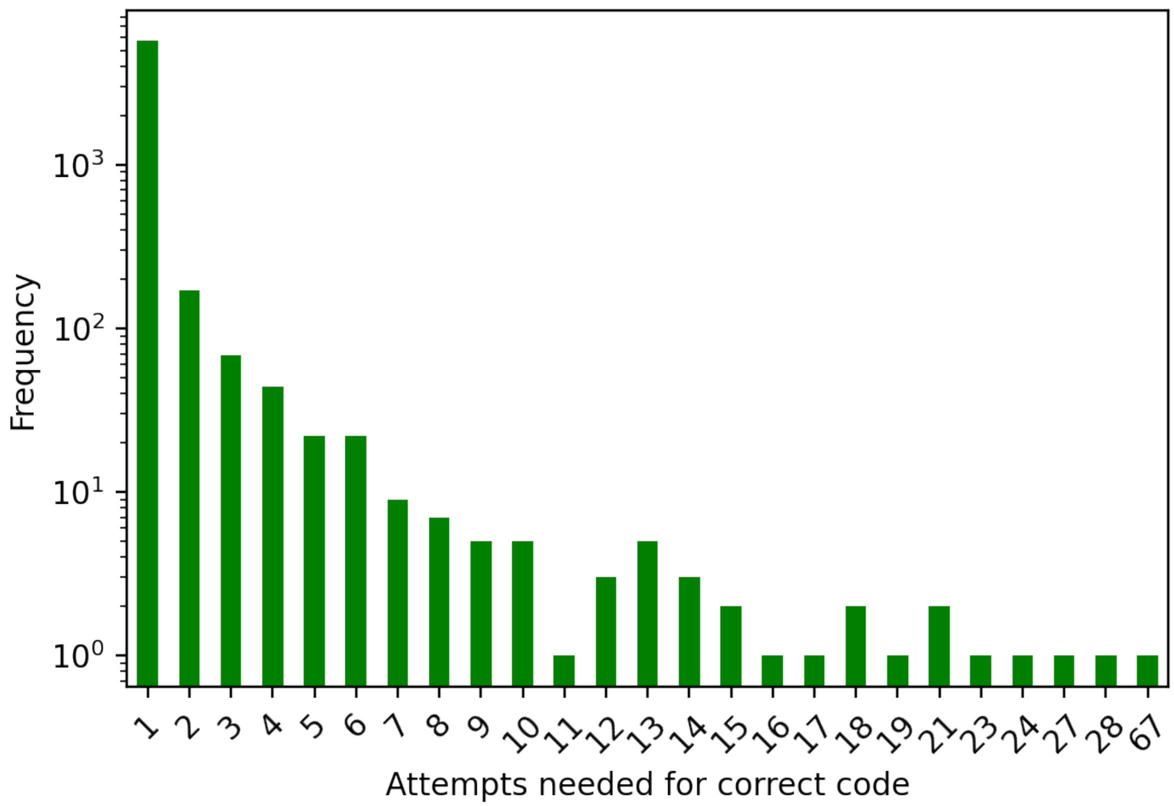

Figure 25: Steps to success (original implementation)

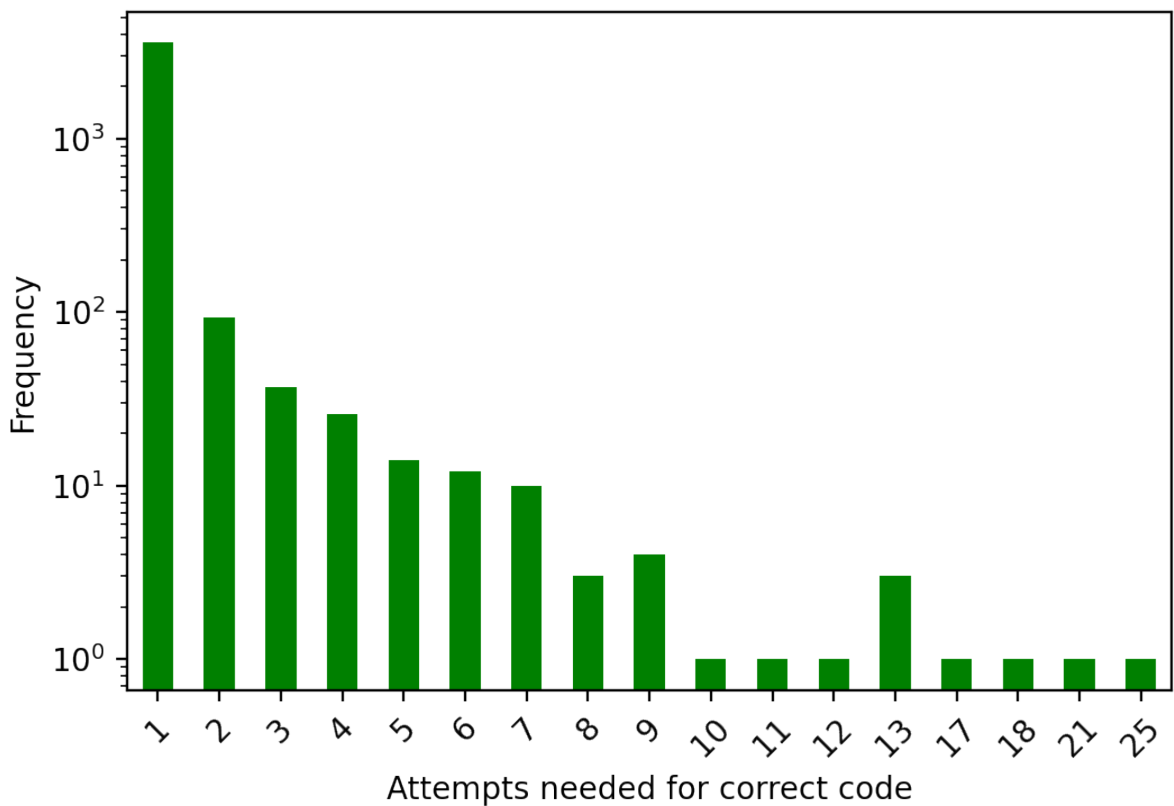

Figure 26: Steps to success (GFM implementation)

7.3 Usefulness analysis

In this subsection, a usefulness analysis is performed on the data gathered on the GFM implementation. Notice that the data on the original implementation gathered on the A/B testing is not used in this section, as the main focus is on the GFM implementation. As explained earlier in Section 6 we keep track on how the user interacts with the model. Most important are the user interactions with the feedback box and the answers on the level-dependent questions. Both are stored in an array with values being either *True*, *False* or *None*. For the array of interaction: *True* when expanded by the user, *False* when not expanded and *None* when the level of feedback hasn't been reached. And for the feedback questions: *True* when the user answers *Yes* on the level-dependent question, *False* when answering *No* and *None* when the level of feedback hasn't been reached. Lastly, a "-" is stored when the user decides to close the question window.

First we look at the *interaction rate*. The interaction rate is divided in two parts: First, we analyse if a unique session has had any interaction with the model at all (expanded any level of feedback). Then we focus on the interaction rate per feedback level of the model. As explained in Section 6 the title of the feedback box is level-dependent, already giving the user information on the expected information before expanding the box. For both, a different analysis is performed between the general percentage of interactions and the percentage of interactions only looking at unique users. A large difference in these numbers would suggest that the model has less added value when being interacted with multiple times, making the users no longer expand the feedback box. Then we analyse the feedback given by the user on the level-dependent questions. Giving insight in the usefulness of the different levels of the model. All are visualized using bar plots. Next to the level-dependent questions, the general feedback question is analysed and visualized as well. Followed by an analysis of the copying behaviour of the user. We compare the code submission of a unique session before and after being presented with the *similar code* level of feedback to analyse the code-changing behaviour.

7.3.1 GFM interaction rate

In this section, we analyse the user interaction rate with the model. We analyse the interaction of the user with the model in general, as well as the interaction with each level of feedback. First, we look at the general interaction with the model. Interaction is stated as *having expanded one or more levels of feedback*. So, if any level of feedback is expanded by the user, we count this as interaction. We filter on all situations where the model has been presented to the user. This is the case when the array of Booleans of the *collapse* value contains either a *True* or *False* value. For a total of 85 sessions 41 had interaction with the model while 44 hadn't. Respectively, 48.2% and 51.8%. A visualization of the results can be found in Figure 27. Notice that we don't filter on unique sessions, and the value of 85 sessions is not the same as the unique amount of users. We are also interested if a unique session (or user) has had any interaction with the model at all, for example when the model being presented multiple times. Therefore, we iterate over the different sessions and analyse if any of the *collapse* value is *True*, in that case we know the user interacted with one or more levels of feedback. For a total of 48 unique sessions 26 had interaction with the model while 22 hadn't. Respectively, 54.2% and 45.8%. This interaction rate is visualized as well and can be found in Figure 28. There is a slightly higher percentage in interaction rate when focusing on unique sessions. However, this is not significant and the expected decrease in interaction rate over time is not found.

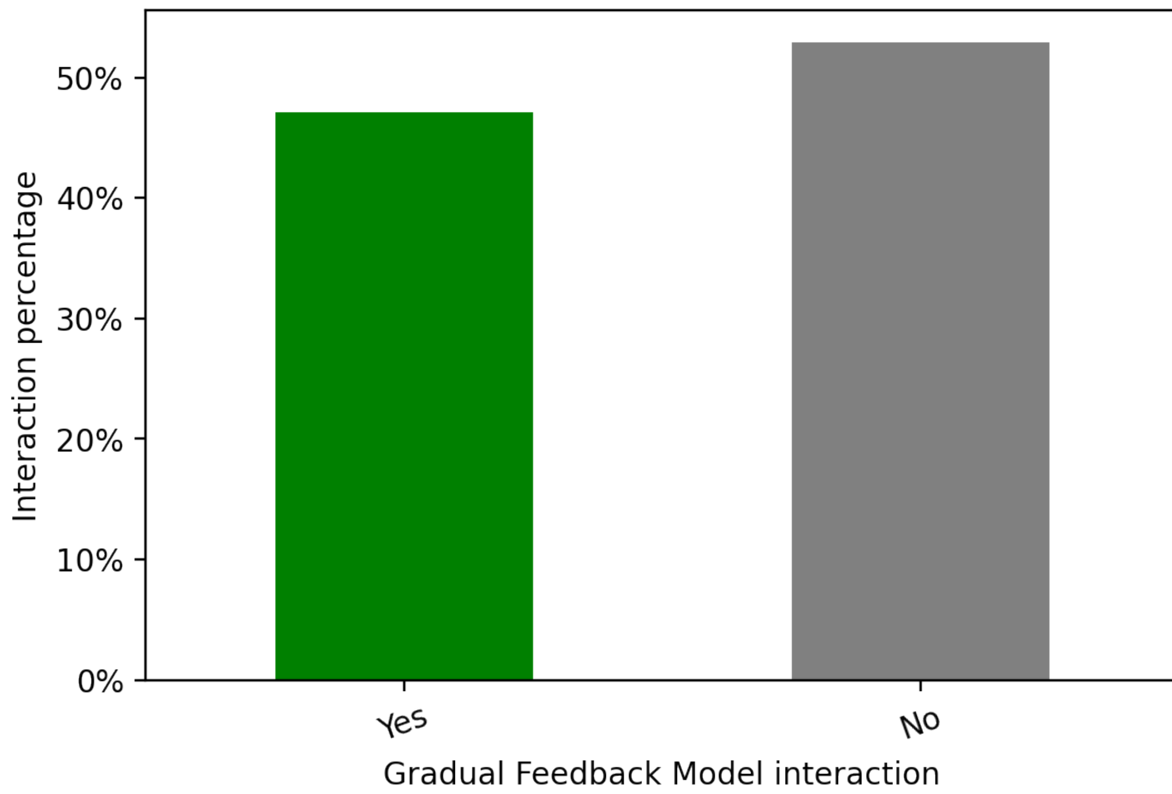

Figure 27: GFM overall interaction rate

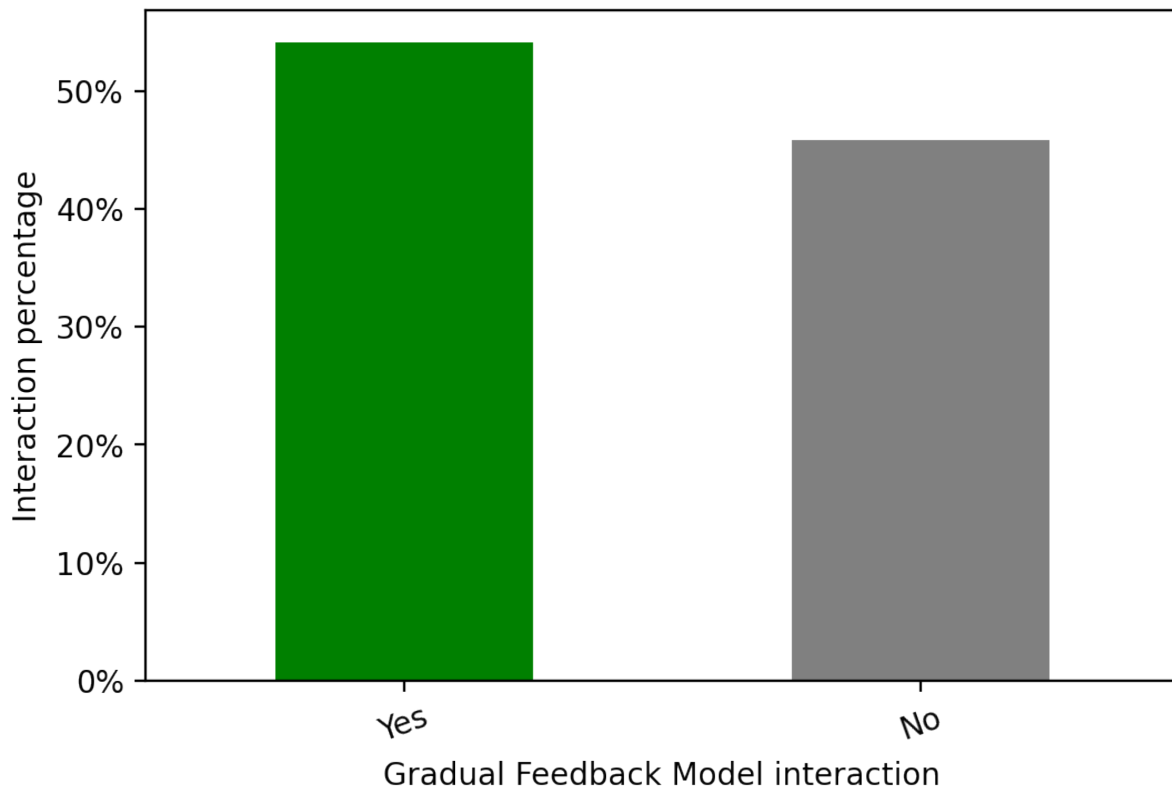

Figure 28: GFM unique overall interaction rate

Next, we look at the interaction rate at each specific level of feedback. On the front-end implementation, each level of feedback has a unique title on the feedback box, already giving the user information on the expected help in the box. Titles are short but explanatory, such as *similar code* or *suggestion*. A level of feedback is returned to the user, a total of 164 times for which they expanded the window 51 and 113 times they didn't. This interaction rate is lower than expected, especially on the *higher* feedback levels. The cause is unknown and should be further researched in future work. An overview of all levels can be found in Table 24. Notice that in this case we look at the total amount of model interactions, not only at unique session. For example, the similar code level has been shown 40 times, but only to 20 unique sessions.

In Figure 29 the interaction rate is plotted for each level of feedback. We see that interaction rate is the highest for the first level of feedback, something that is expected. Surprisingly, the interaction rate of the *similar code* level is around 25%. This was expected to be (far) higher. Similar to the general interaction rate, we do an additional analysis for *unique* sessions, analysing if the user has had any interaction with the specific levels of feedback. For example, when they decided to not expand the window on the first interaction possibility but did so on the second one. These numbers are visualized as well and can be found in Figure 30. There seems to be a slightly higher percentage on interaction, but not significant. Which supports the earlier stated conclusion that users interested in the model keep interacting with it, while users that aren't interested will keep their same behaviour. An overview of the numbers can be found in Table 25.

Feedback Level	No	Yes
Expanded Error	50	35
Similar Code	31	9
New Concepts	21	4
Break Suggestion	11	3

Table 24: GFM feedback levels interaction rate (total)

Feedback Level	No	Yes
Expanded Error	24	24
Similar Code	20	7
New Concepts	16	4
Break Suggestion	9	3

Table 25: GFM feedback levels interaction rate (per unique session)

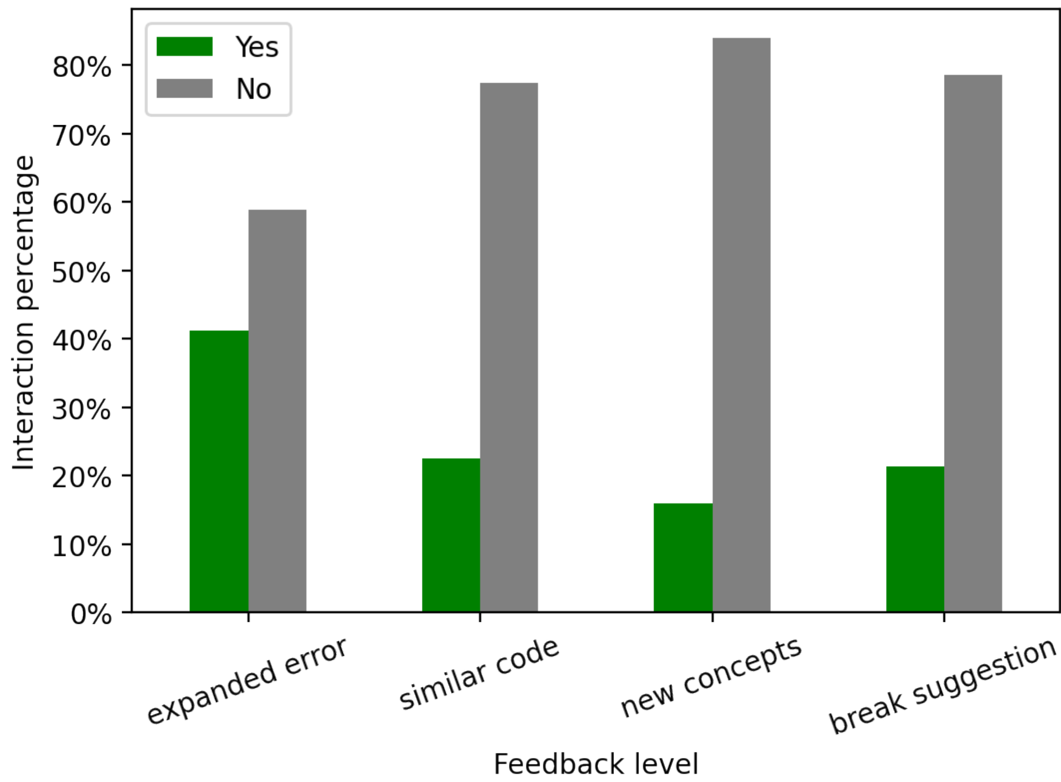

Figure 29: GFM interaction rate for each feedback level

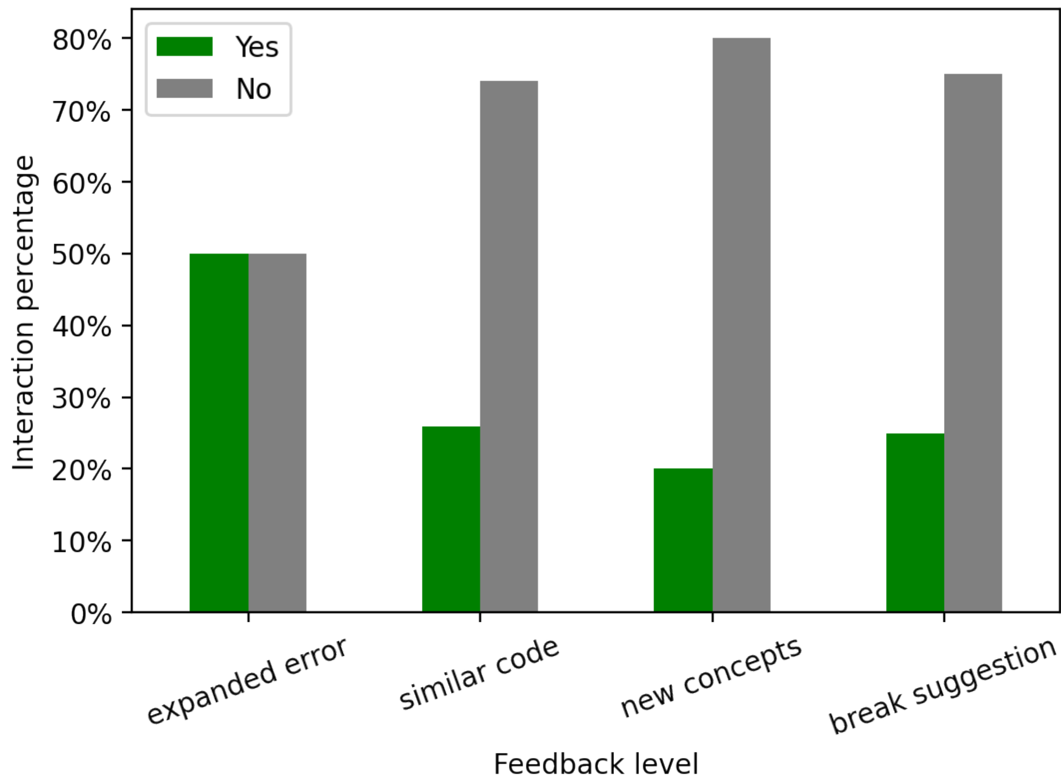

Figure 30: GFM unique interaction rate for each feedback level

7.3.2 Feedback usefulness

In this section, we analyse the user feedback on the yes/no questions asked when a correct program is created after making consecutive mistakes. The questions are only asked if the users interacted with the level specific for that questions. The general feedback is always asked if any interacted has occurred at all. First, we look at the answers to the general feedback question, in total as well as per unique session. Then we analyse the answers to the level-specific questions. Similarly, we also look at the total and the answers per unique session. With the analyses on unique sessions, we analyse if the user did perceive the level as useful at least once. So, for example, when being presented the level of feedback four times, three times answering *False* and one time *True*. In the case when only *False* and "-" are answered, *False* is choosing as the user's answer. The unique usefulness will still be counted as *True*.

The general feedback question is asked a total of 41 times, for which 17 are answered with *True*, 16 with *False* and 8 users closed the question window (automatically answering "-"). In 45 cases, none of the levels of feedback is expanded, so the general feedback question is not asked, as there hasn't been any user-model interaction. These numbers should be corresponding with the interaction rate analysis, but for unknown reasons we find 45 cases of *None* instead of the expected 44. The exact numbers in total as per unique session can be found in Tables 26 and 27. The visualization calculated as a percentage of total answers per feedback level can be found in Figures 31 and 32. Notice that the general question is not included in the visualization. The found usefulness rate is slightly higher for overall interaction per unique sessions than in total, suggesting that the model has diminishing usefulness over time. However, this difference is not significant, and no clear conclusion can be drawn without further research.

Feedback Level	Yes	No	-
General Question	17	16	8
Expanded Error	14	15	7
Similar Code	4	3	2
New Concepts	5	0	0
Break Suggestion	1	3	0

Table 26: GFM feedback levels usefulness rate (total)

Feedback Level	Yes	No	-
General Question	14	8	4
Expanded Error	13	8	3
Similar Code	3	2	2
New Concepts	4	0	0
Break Suggestion	1	2	0

Table 27: GFM feedback levels usefulness rate (per unique user)

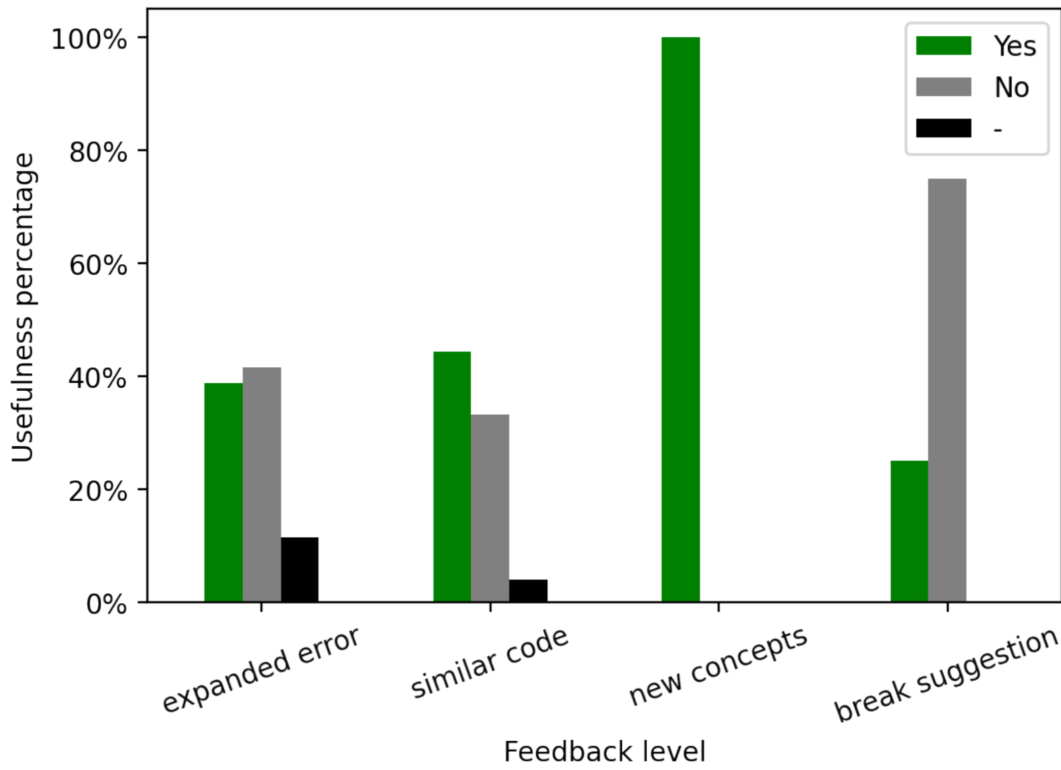

Figure 31: GFM usefulness rate for each feedback level

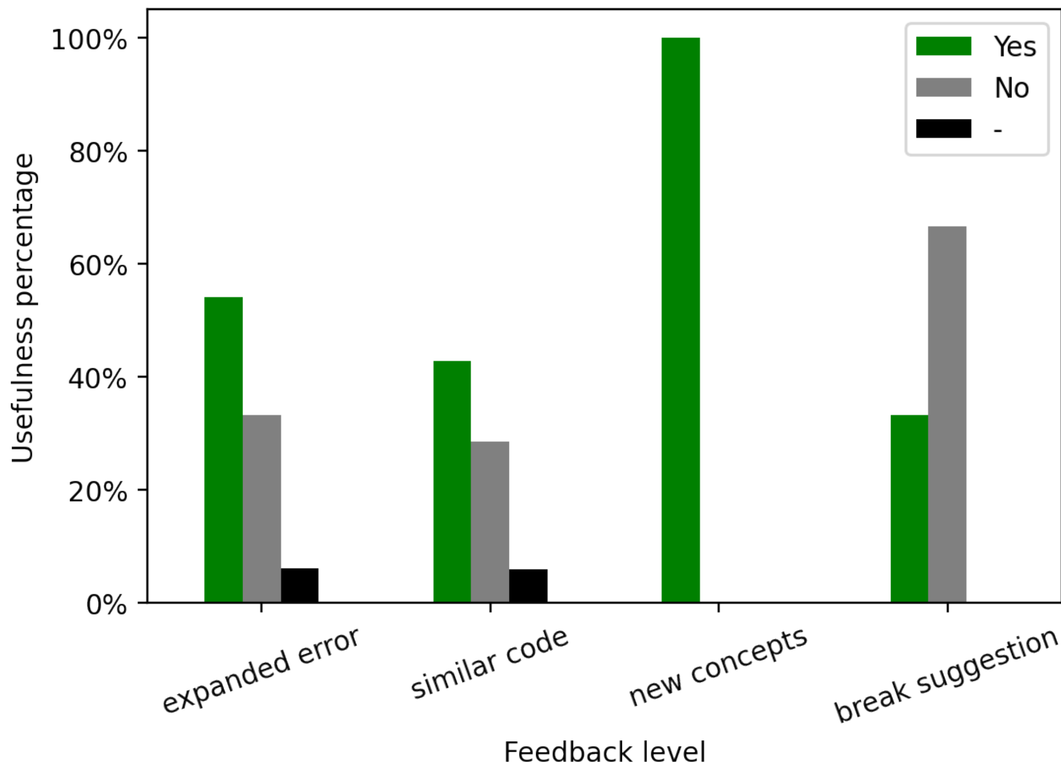

Figure 32: GFM unique usefulness rate for each feedback level

7.3.3 Copying analysis

In this section, we analyse the copying behaviour of users after having interaction with the similar code level of feedback. The (faulty) code being submitted before the similar code interaction is compared to the similar code. Followed by comparing the submitted code and suggested similar code after the interaction with the feedback level. An increase in similarity is expected, indicating that the users copy or at least get inspiration from the suggested code. Due to the complexity of the task at hand, the filtering and selecting of code is performed manually.

First, all submissions containing a value for the *similar_code* value in the dataset are filtered. Then for each of these sessions they are manually tracked to compare the code submission on feedback level 3 (the similar code is received) and feedback level 4 (following code submission). To improve the analysis, only users with interaction on the similar code feedback level are analysed. In the case of not expanding the feedback box similar code is still calculated, stored and returned, but the users didn't have any interaction. Performing a copying analysis wouldn't make any sense in this case. A total of 20 sessions reached feedback level 3 for which 9 interacted with the model and 11 didn't. 7 of these sessions were unique. For each of these 7 sessions, we manually retrieve the code submissions and analyse the code submissions. In most cases the similar code does not seem to have influence on the code-changing of the user, most error where small and solved by adding a comma or correcting a misspelled keyword. It is unclear if the similar code did help the user or that the user simply spotted the mistake themselves. The errors are too small to determine the behaviour from data analysis alone. We look closely at one user to look more in-depth at the code-changing behaviour.

One (anonymous) user is struggling to execute a correct Hedy program on level 2. An initial attempt is made to ask the user for the name of their dog and then print this name. But the user created a variable consisting of two words, something that is not allowed in Hedy. The restriction that variable names are not allowed to have spaces is unclear to the user, and the error messages do not help solve the problem. An error message is returned stating that the last character at line number 1 is not allowed because it's a punctuation mark. The user replaces the *?* by a *.* character, but still the same error is returned. This behaviour is not the fault of the user, as the original error message is solved correctly; removing the faulty character. The mistake is in the error message, that returns a wrong message due to not correctly catching the two-words-variable mistake on the next line. After removing all punctuation marks at line number 1 a new error is return that the word *dogs* is not a command in Hedy level 2. Closer to the actual problem, but still not helping the user. The returned similar code, which has a one word variable, does not help the user with the problem because he is not understanding the mistake at hand. We argue that similar code is only useful when the user *understands* the problem but can't find it, but does not provide useful information when the current error is not understood, as nothing is learning from copying an example. Similarly, all additional feedback returned to the user is not useful, as the original mistake and corresponding error messages are unclear and confusing to the user. Returning extra information (that does not suggest a solution either) does not help to solve the error. An overview of the code-changing behaviour of this user can be found in Code Snippets 4, 5, 6 and 8.

```
dogs name is ask What is your dogs name?
print dogs name is your dogs name!

The code you entered is not valid Hedy code.
There is a mistake on line 1, at position 40.
You typed a question mark, but that is not allowed.
```

Code 4: First attempt and corresponding error message

```
dogs name is ask What is your dogs name.
print dogs name is your dogs name!

The code you entered is not valid Hedy code.
There is a mistake on line 1, at position 40.
You typed a period, but that is not allowed.
```

Code 5: Second attempt and corresponding error message

```
dogs name is ask What is your dogs name
print dogs name is your dogs name!

dogs is not a Hedy level 2 command. Did you mean is?
```

Code 6: Third attempt and corresponding error message

```
name is ask what are you doing name?
print name is what you are doing?
```

Code 7: Similar (correct) code returned to user

```
color is ask What is your favorite color?
print color is your favorite!
```

Code 8: User's final code submission

8 Discussion

In this section, we discuss the results of the Gradual Feedback Model implementation and the corresponding A/B testing. First we discuss the general results such as the error rate and drop-out rate. Then we look at the overall results of the model implementation and discuss unexpected findings and their possible explanations. Finally, we discuss the limitations of the model implementation and how they could have influenced the results.

8.1 Results

While we find a decrease in error rate and through statistical analysis conclude that this is indeed significant, we are unable to support this finding by our other analysis. For unknown reasons, we find a significant increase in drop-out rate between the original and the GFM implementation. A feature that we state as an important indicator for error message understanding. It might be that our definition of a *drop-out* is too broad, and further work should investigate further on classifying a drop-out. We also find no indication that the GFM implementation helps users solve their errors faster, as the *steps to success* analysis indicate no difference between both implementations. The higher findings for the original implementation can be seen as outliers. The total model interaction rate is slower than expected, being slightly higher than 54% on unique user interaction and slightly higher than 48% on total amount of interactions. Notice that this is classified as having interaction with *at least* one level of feedback before running correct code. The interaction rate per level shows a similar pattern, being higher for unique users and decreasing over time with the total amount of interactions. Especially the low interaction rate on the *similar code* level is unexpected and future work should aim at better understanding the behaviour patterns of users when error solving. Similarly, the model usefulness follows an almost identical pattern: being higher for unique users and decreasing over interactions. Suggesting that the users perceive the model as less useful when being presented with it multiple times.

8.2 Limitations

Several limitations are found and/or introduced in the model and the implementation within Hedy. The first limitation is the linear implementation of the similar code finding. Due to the structure, identical correct code is proposed when a user makes a mistake with the same *keyword structure*. When a programmer is stuck at a program and does not make significant changes between faulty submissions, the chances are they receive the same, similar code. Which is not useful, because it didn't help them solve the problem the first time. Another limitation on the implementation is the lack of model-user interaction data. Only when consecutive faulty submissions are followed by a correct submission the feedback questions on the model are asked. Other than the *collapse* value, no other model-user interaction data is gathered and/or stored. This complicates the process of data analysis and the conclusions drawn. One might also be interested in the time between a faulty submission and the expanding of the feedback box. For which we can conclude if the user first reads the original error message or directly expands the feedback box and reads the additional one generated by the model. While this would be difficult to implement within Hedy due to the current code structure, it is still a limitation in the research.

9 Future Work

In this section, possible future work is discussed on the research of this Thesis. Several small suggestions were already made throughout the Thesis, where the different design and implementation choices were explained. As well as the discussion on the results and the limitations. However, in this section, a differentiation is made between three large possible sections of research to further improve the Gradual Feedback Model and/or the implementation.

9.1 Improving the implementation

One large aspect of improvement would be the *feedback logging*. Currently, the interaction with and feedback on the model are only logged *after* a successful code submission. So if, after some mistakes, no correct submission is made, the interaction from the user with the model is never logged. This is due to the structure of the Hedy web environment. Currently, a first logging is made *server-side* right after the parsing, whereas the interaction logging should be made after the user has received and interacted with the website. Due to this difficulty, the choice was made to temporary store the interactions between the user and the model and make a GFM logging once: after a correct code submission. While this is a good choice from the server-load perspective, useful data on interaction is lost due to a user quitting before a correct submission. This aspect can and should be improved when aiming to improve the GFM implementation within Hedy.

9.2 Increasing usefulness

As found in the data analysis of the user feedback on the *Gradual Feedback Model*, there is a difference per feedback level on the user's perception of usefulness and the interaction rate is lower than expected. Due to COVID-restrictions, we were unable to perform any observational studies on novice programmers to better understand their error solving behaviour and error message needs. All insights were found through similar work or data analysis on the Hedy dataset. An observational user study should be performed to better understand what user perceive as useful error messages and how to improve these. The observational study can be conducted within a primary school classroom combined with interviews when using the Hedy implementation, or in any Introduction to Programming course on Python when creating a Python implementation of the Gradual Feedback Model.

9.3 Language-independent implementation

As explained in Section 5.2 the concept of the model is suitable for any textual programming language. Due to the small syntax possibility of Hedy and the corresponding relatively small programs, the additional feedback messages might not be as useful as expected. It should be interesting as future work to implement the model within Python to analyse the usefulness on a programming language with more programming syntax and possibilities. It should be noted that the feedback level of *new concepts* does not apply for non-gradual languages, as all programming concepts and syntax are available from the starting point. The model should be altered at this level, while all other level are language-independent. There is no suggestion on an alternative for this feedback level, and it is advised to research this in combination with the observational study to get more insights in the feedback needs of the user.

10 Conclusion

In this section, we conclude on the work done. The data analysis on over 1 million Hedy programs gave useful insights in the mind and behaviour of novice programmers. While the in-development status of the Hedy programming language complicated the process, levels and structures kept changing over time, a good general view could be created. Combined with the literature review on similar work and the literature study on other research in the field of program analysis, a good foundation was created for error message improvement. The proposed model: the *Gradual Feedback Model* has been implemented within the Hedy web environment and tested through A/B testing.

We find a statistical significance on the decrease in error rate with a confidence interval of over 95%. For which, we can conclude that the model does indeed help decrease the errors made by novice programmers. No decrease (but instead an increase) in drop-out rate is found, as well as no significant difference is found in the *steps to success*. Concluding that other than the significant decrease in error rate, there are no clear indications on the usefulness of the Gradual Feedback Model. We found an interaction and usefulness rate, which are both lower than expected. Through a copying analysis, we found no indication that users use the similar code as a debugging strategy and copy code to solve the error at hand.

Future work should be conducted to improve the implementation of the model within Hedy. By keeping tracking of the user-model interaction and gather more useful data to better understand the user. To improve the model itself, an observational study should be conducted followed with one-on-one interviews to better understand the user needs on error messages and additional help throughout the debugging process. Lastly, the model should be implemented in another textual programming languages such as Python to compare the interaction and usefulness of the model between Hedy and a programming language with more complexity and a larger syntax library.

References

[1] Alan Agresti, Barbara Finlay, et al. "Statistical methods for the social sciences". In: (2009).

[2] Amjad Altadmri and Neil CC Brown. "37 million compilations: Investigating novice programming mistakes in large-scale student data". In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. 2015, pp. 522–527.

[3] T.B. Bakker. *Hedy: Implementing the Gradual Feedback Model*. `https://github.com/TiBiBa/hedy`. 2021.

[4] Raul Barbosa et al. "The most frequent programming mistakes that cause software vulnerabilities". In: *arXiv preprint arXiv:1912.01948* (2019).

[5] Neil CC Brown and Amjad Altadmri. "Investigating novice programming mistakes: Educator beliefs vs. student data". In: *Proceedings of the tenth annual conference on International computing education research*. 2014, pp. 43–50.

[6] Neil CC Brown and Amjad Altadmri. "Novice Java programming mistakes: Large-scale data vs. educator beliefs". In: *ACM Transactions on Computing Education (TOCE)* 17.2 (2017), pp. 1–21.

[7] Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. "Enhancing syntax error messages appears ineffectual". In: *Proceedings of the 2014 conference on Innovation & technology in computer science education*. 2014, pp. 273–278.

[8] Björn Hartmann et al. "What would other programmers do: suggesting solutions to error messages". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2010, pp. 1019–1028.

[9] Bart Heemskerk. "The Error that is the Error Message: Comparing information expectations of novice programmers against the information in Python error messages". In: (2020).

[10] Felienne Hermans. *Hedy*. `https://github.com/Felienne/hedy`. 2021.

[11] Felienne Hermans. *Hedy*. `https://www.hedycode.com/`.

[12] Felienne Hermans. "Hedy: A Gradual Language for Programming Education". In: *Proceedings of the 2020 ACM Conference on International Computing Education Research*. 2020, pp. 259–270.

[13] Antonio Santos Júnior, Jorge César Abrantes de Figueiredo, and Dalton Serey. "Analyzing the Impact of Programming Mistakes on Students' Programming Abilities". In: *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*. Vol. 30. 1. 2019, p. 369.

[14] Monika Kaczorowska. "Analysis of typical programming mistakes made by first and second year IT students". In: *Journal of Computer Sciences Institute* 15 (2020).

[15] Tobias Kohn. "The error behind the message: Finding the cause of error messages in python". In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 2019, pp. 524–530.

[16] Michael Kölling et al. "The BlueJ system and its pedagogy". In: *Computer Science Education* 13.4 (2003), pp. 249–268.

[17] Anita Krishnamurthi, Ron Ottinger, and Tessie Topol. "STEM learning in afterschool and summer programming: An essential strategy for STEM education reform". In: *Expanding Minds and Opportunities* (2013), p. 31.

[18] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. "Measuring the effectiveness of error messages designed for novice programmers". In: *Proceedings of the 42nd ACM technical symposium on Computer science education*. 2011, pp. 499–504.

[19] Roy D Pea. "Language-independent conceptual "bugs" in novice programming". In: *Journal of educational computing research* 2.1 (1986), pp. 25–36.

[20] Raymond S Pettit, John Homer, and Roger Gee. "Do Enhanced Compiler Error Messages Help Students? Results Inconclusive." In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 2017, pp. 465–470.

[21] James Prather et al. "On novices' interaction with compiler error messages: A human factors approach". In: *Proceedings of the 2017 ACM Conference on International Computing Education Research*. 2017, pp. 74–82.

[22] Rebecca Smith and Scott Rixner. "The error landscape: Characterizing the mistakes of novice programmers". In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 2019, pp. 538–544.

[23] John Stachel et al. "Managing cognitive load in introductory programming courses: A cognitive aware scaffolding tool". In: *Journal of Integrated Design and Process Science* 17.1 (2013), pp. 37–54.

[24] Emillie Thiselton and Christoph Treude. "Enhancing python compiler error messages via stack". In: *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE. 2019, pp. 1–12.

[25] V Javier Traver. "On compiler error messages: what they say and what they mean". In: *Advances in Human-Computer Interaction* 2010 (2010).

[26] Christopher Watson, Frederick WB Li, and Jamie L Godwin. "Bluefix: Using crowd-sourced feedback to support programming students in error diagnosis and repair". In: *International Conference on Web-Based Learning*. Springer. 2012, pp. 228–239.

11 Appendix A: GFM Hedy Messages

In this Appendix, all GFM messages implemented within the Hedy environment are shown in different tables. Notice that their correlation with the Hedy implementation is not shown, and the tables are only here for completeness of the implementation discussion. For a complete overview of their connection with the original Hedy error handling, one should look at the open-source project on GitHub. [3] All messages can be found in Tables 28, 29, 30 and 31.

Error name	Message
Expanded_Wrong Level	Remember, Hedy is a gradual programming language. That means that the commands can differ for each level.
Expanded_Incomplete	Your code is incomplete, try to finish it with the error message above.
Expanded_Invalid	You used a command that doesn't exist. Take a closer look if you haven't made a mistake by accident or used a non-alphabetical letter at a weird spot.
Expanded_Invalid Space	Just like with normal text we don't want to start with a whitespace, put your code at the most-left side of the screen.
Expanded_Parse	You're not allowed to use all characters at every spot, then the computer won't understand! Try to solve the error with the message above.
Expanded_Unquoted Text	You forgot a quote somewhere. Remember that each sentence you want to print should start and end with a quotation mark.
Expanded_VarUndefined	You can't print a variable if it doesn't exit. Do you want to print text? Take a closer look at how to use the print command in this level.
Expanded_Unknown	Computers doesn't always understand it either, take a close look at your code and otherwise try it again.

Table 28: GFM Hedy implementation Enhanced Messages

Error name	Message
Identical_code	If you don't change anything the same error will occur. Remember: computers will always respond the same way!
No_similar_code	No similar code has been found...
Break	You seem to be stuck at this level, take a little break and try it again later.

Table 29: GFM Hedy implementation Additional Messages

Error name	Message
New_level1	Remember, in level 1 we can only use the "print", "ask" and "echo" commands.
New_level2	Remember, the use of a name is new in level 2.
New_level3	Remember, printing something between quotes is new in level 3.
New_level4	Remember, the "if" and "else" commands are new in level 4.
New_level5	Remember, the "repeat" is new in level 5.
New_level6	Remember, the use of "calculations" is new in level 6.
New_level7	Remember, using "indents" is new in level 7.
New_level8	Remember, the "for" command is new in level 8.
New_level9	Remember, The ":" symbol the line before the indent is new in level 9.
New_level10	Remember, using "for" and "if" nested is new in level 10.
New_level11	Remember, brackets around "print" and "input" are new in level 11.
New_level12	Remember, lists work differently in level 12: we know need "square brackets".
New_level13	Remember, for a comparison we use "==" in level 13.

Table 30: GFM Hedy implementation New Concept Messages

Error name	Message
Feedback_general	It works, well done! Did you perceive the additional information (in the blue box) as useful?
Feedback_question2	Did the extended explanation help you to solve the error?
Feedback_question3	Did the similar code help you to solve the error?
Feedback_question4	Did the recap on the new elements help you to solve the error?
Feedback_question5	Did the break suggestion help you to solve the error?

Table 31: GFM Hedy implementation Feedback Questions