The Future of Engineering Education
2024 Annual Conference & Exposition
Oregon Convention Center ASEE
Portland, OR . June 23 - 26, 2024
Paper ID #41184

An Initial Investigation of Design Cohesion as a IDE-based Learning Analytic for Measuring Introductory Programming Metacognition

Dr. Phyllis J. Beck, Mississippi State University

Phyllis Beck is a blend of art and science having completed an undergraduate degree in Fine Arts at MSU and a Ph.D. in Computer Science, where she focused on applying Artificial Intelligence, Natural language Processing, and Machine Learning techniques to the engineering education space. Currently, she is working as an Assistant Research Professor at Mississippi State University in the Bagley College of Electrical and Computer Engineering. She has worked for companies such as the Air Force Research Laboratory in conjunction with Oak Ridge National Labs and as an R & D Computer Science Inter for Sandia National Labs conducting Natural Language Processing and AI research and was inducted into the Bagley College of Engineering Hall of Fame in 2021.

Dr. Mahnas Jean Mohammadi-Aragh, Mississippi State University

Jean Mohammadi-Aragh is the Director of Bagley College of Engineering Office of Inclusive Excellence and Associate Professor in the Department of Electrical and Computer Engineering at Mississippi State University. Through her interdependent roles in research, teaching, and service, Jean is actively breaking down academic and social barriers to foster an environment where diverse and creative people are successful in the pursuit of engineering and computing degrees. Jean's efforts have been recognized with numerous awards including the National Science Foundation Faculty Early Career Development award, the American Society for Engineering Education John A. Curtis Lecturer award, and the Bagley College of Engineering Service award. Jean earned her B.S. and M.S. in computer engineering from Mississippi State University, and her Ph.D. in engineering education from Virginia Tech.

An Initial Investigation of Design Cohesion as an IDE-based Learning Analytic for Measuring Introductory Programming Metacognition

In this full paper, we describe our case study approach to initially characterize Design Cohesion as a new IDE-based learning metrics through an exploratory coding process. We developed a new alignment notation to generate two new qualitative metrics: *Design Cohesion* (High, Medium, Low) and *Granularity Level* (High, Medium, Low) *Design Cohesion* is the level of alignment between flowchart and code, which accounts for the order of intended program execution, the internal data of a flowchart node, and the number of nodes in the flowchart that map to the code. We define *Granularity Level* as an additional characterization of cohesion that labels the level of detail in the flowchart. Our primary objective is to use these new metrics to understand how a flowchart can be aligned with its code implementation to understand introductory students' current level of programming metacognition.

In the context of our case study, we discuss the exploratory coding process and alignment notation developed to generate the features for the newly proposed metrics. Next, we explore two cases to illustrate the diversity of characteristics found in various feature combinations. Each case study compares two examples from the same participant, one with High Cohesion and High Granularity, the other with High Cohesion and Low Granularity.

Our initial investigation into design cohesion has led to the hypothesis that High-level Design Cohesion paired with low levels of flowchart granularity demonstrates high levels of abstraction in the initial flowchart design, which may point to under-designing by participants and/or lower levels of metacognition. Comparatively, having high cohesion and granularity may point to over-designing by the participant and often stems from a one-to-one mapping of flowchart nodes to lines of code. Our results point toward a logical relationship between Design Cohesion and students' level of self-estimated skill, and we are confident that Design Cohesion will serve as viable metric for understanding introductory programming metacognition.

1. Introduction

This paper presents our initial characterization of *Design Cohesion* and *Granularity Level* and our case study approach to the qualitative exploratory coding process applied to the flowchart and programming data collected as a exploratory study across 25 participants and 50 code samples. We seek to explore these metrics as candidates for our five-dimensional model of programming skill estimation and to determine if they are effective methods for characterizing introductory programming metacognition. The *Design Cohesion* metric is an alignment metric that proposes to classify the level of accuracy between a flowchart design and the code implementation of a programming problem. It aims to accounts for the order of intended program execution, the internal data of a flowchart node, and the number of nodes in the flowchart that map to the code. The *Granularity Level* further characterizes design cohesion by classifying the level of detail contained within a flowchart compared to the final program solution.

Our research is motivated by a primary goal: the development of a model of programming skill estimation for introductory programming grounded in effective learning theories. By applying artificial intelligence and IDE-based learning analytics for collecting and analyzing programming process data, we seek to provide real-time personalized metacognitive feedback to introductory programming students on their current skill level.

By combining learning analytics methods with instrumented IDEs, we have the opportunity to remove the labor-intensive task of providing individualized feedback to students on their programming process. However, current IDEs (Integrated Development Environments) lack the metrics to assess strategic knowledge effectively. Programming evaluation and teaching methods seem to focus entirely on the syntactic and procedural level instead of the strategic level despite the research showing that directly teaching metacognitive strategies and problem-solving techniques improves student outcomes [10, 12, 13].

The design phase of a program solution, though encouraged, is almost entirely ignored and, as a result, not captured within IDE-based environments or learning analytics platforms. In order to begin assessing strategic knowledge, we need new methods to capture programming metacognition. This includes the development of more meaningful metrics for identifying metacognitive strategies. These metrics then need to be situated within a framework that relates the evaluation of syntactic and procedural knowledge to the level of strategic knowledge. By understanding students' approach to the design of a solution, we can determine if the difficulties in completing a programming problem lie in conceptual understanding of the problem and general programming constructs or if the issue lies with the programming language itself and the ability to translate a design into a concrete programming solution.

Our approach to providing additional insight into student's capacity to design is to develop Design Cohesion as a new metric for understanding programming metacognition for introductory programming using a custom web-based IDE as part of a broader study to generate new metrics for IDE-based learning analytics. This study consisted of a series of surveys to understand active programming knowledge and a set of programming exercises completed within our custom IDE in a supervised lab environment.

1.1 Research Questions

This qualitative exploratory study seeks to answer the following research question: What metrics can be developed for assessing students' level of design cohesion based on the alignment between flowchart design and source code implementation of a programming exercise?

Additionally, we aimed to collect data that demonstrates the web based IDE as a proof of concept for collecting programming processing data and if design cohesion is a viable IDE-based learning analytic for programming metacognition.

The following sections lead with a discussion on the relevant background and related works on IDE-based learning analytics and flowcharts in programming education. We then discuss the methods for our study design and participant recruitment. This is followed by a detailed walkthrough on the development of the alignment notation used to code the flowchart and programming data to assist in the development for the codebook on design cohesion and the classification process of each sample. We then present the codebook that was formulated as a result of the coding process for Design Cohesion and Granularity Level. To further clarify the application of this process we provide two case studies and close with a discussion on the results, current limitations and future work.

2. Background and Related Works

For clarity and context the following section provides a brief overview of the relevant literature and related work which touches on IDE-based learning analytics, a historic overview of flowcharts in programming and a description of our model of programming skill estimation.

2.1 IDE-based Learning Analytics

Currently, students still face many challenges when learning to program, and instructors struggle with the time-intensive process required to give students the appropriate level of feedback to excel. These challenges have increased the need for automated tools and methods to gather and analyze programming artifacts and understand student programming patterns in introductory computing courses. Research indicates that students lack the problem-solving skills to apply strategic knowledge to novel problems in real-world contexts. These challenges have led to a need to automate methods to identify what programming concepts students struggle with, identify at-risk students, and understand what metacognitive strategies and patterns students use to solve programming problems [5, 12, 13]. By developing methods for early identification, we can give instructors time to intervene and provide the appropriate level of feedback, improving a student's ability to succeed during the most formative times in their computing education [6, 8].

There have been many attempts at developing novel approaches to support various aspects of programming metacognition, improve self-efficacy, and provide automated feedback and assessment for students in introductory programming courses [5, 6, 8]. *Programming metacognition* can be broadly defined as how students think about programming and the problem-solving strategies they employ to achieve a goal when given a programming task [9]. However, most of these methods have yet to be successfully scaled and applied in the classroom. Previous studies suffer from issues such as being too small, difficult to validate or replicate, and software that is not shared or is abandoned. Software developed as part of research efforts is typically abandoned due to a lack of research funding or the high overhead of developing a system to collect and analyze data on a large scale [6, 7, 8]. Despite the success of some methods, advances have not been integrated into a cohesive platform where instructors can automate the data acquisition process [7, 6]. Furthermore, no platform can capture a persistent trace of the complete programming process, including the design phase.

Programming platforms and Integrated Development Environments (IDEs) also lack meaningful feedback and skill estimation and still require extensive time from instructors to provide individualized feedback. Typically, a programming assignment is graded on whether or not the program generates the correct output. Little feedback is given on how the problem was solved or how the code was structured. However, code is read more frequently than written, and the ability to effectively debug, test, document, and write well-structured and maintainable code is a critical professional skill. Unfortunately, these skills are only briefly discussed in a typical computer science course [10]. Students are rarely evaluated on them, leading students to develop poor habits that follow them into their professional careers. Additionally, educators cannot see the effects of their interventions as they are affecting their classrooms in real time [6, 8]. This leaves space for researchers to explore new approaches to instrumented IDEs and learning analytics for programming education.

2.2 Flowcharts and Programming Education
Historically, flowcharts have been a widely accepted universal representation within the programming process to communicate algorithmic design [19] without the complexity or overhead of a programming language. From their introduction in the 1940s to the development of the ANSI standards [19, 22] and integration into a wide variety of other fields [22], flowcharts have been used within programming as a method for reducing cognitive load. There have been multiple attempts at reducing the cognitive load within programming with the development of mini-languages, Iconic Programming Languages, Pseudo-code, and Block based systems [17, 18]. Current research efforts [16] have shown that flowcharts are most useful when executed at the proper granularity, which helps provide a higher-level overview of a program to assist with metacognition, problem-solving, and the design process. While there is still some debate about the utility of flowcharts within the programming process, there is significantly more evidence for than against them. Studies continue to find improvements in various areas such as academic achievement, self-efficacy, computational thinking, critical thinking, problem-solving, mental simulation, problem decomposition, and code tracing. As students become more familiar with the process and tools of flowcharting and as programming problems grow in complexity, the benefits of flowcharts become more apparent. A recent and effective use case is that of an offline educational tool to prevent students from defaulting to an 'act-first' over a 'think-first' approach to aid in the problem-solving process by teaching one to effectively generate sub-problems and sub-solutions [17]. Overall, flowcharts still present as a promising metacognitive strategy for problem-solving irrespective of programming language.

In spite of the benefits, flowchart and diagramming tools in programming education have had mixed success in being integrated into the classroom with the default being the use of static diagramming tools. Tools range from the development of applications that can generate and execute code from a flowchart such as Raptor and Visual Logic [19], which have largely fallen out of favor due to lack of maintenance, to web-based diagramming tools that only generate static flowcharts such as LucidChart, Cacoo and Dia [20]. Flowgorithm [21] is the only modern cross-platform diagramming application our researchers found that was specifically designed for introductory programming education that can also execute code in multiple languages such as C#, Python, and Javascript. However, currently available tools do not provide a method to capture the full design process, which limits the ability to analyze it beyond the final diagram.

This leaves significant room for developing additional metrics for understanding programming metacognition from students' flowchart and design process.

2.3 The Model of Programming Skill Estimation

The development of Design Cohesion is one of the five dimensions of our current model of programming skill estimation for introductory programming. For additional details and information on each component we refer you to our previous work on this topic.

- *Thinking Processes (TP)* [2, 3, 11, 23]
- *Organizational Strategies (OS)* [1, 3]
- *Design Cohesion (DC)* [3]
- *Skill Mastery (SM)* [3]
- *Timeline of Development* (TD) [3, 4]

Together these five components form a comprehensive model of student programming behavior and problem solving process.

3. Methods

In this section we present our methods for the study design, participant recruitment, and the development and application of the alignment notation. We then present the codebook that was formulated as a result of this coding process and two case studies that further demonstrate the application of the coding process.

3.1 Study Design

The development of Design Cohesion is part of a broader study to generate new metrics for IDE-based learning analytics. This study consisted of a recruitment survey, a prior programming knowledge survey, two programming sessions, each consisting of a single programming problem to be completed in Python and an exit interview. We worked with Alphaflow Labs to develop a web-based IDE to use as the instrument for data collection.

Each programming session consisted of a design phase using a custom web-based drag and drop flowchart diagramming tool and a programming phase with a custom IDE. Programming process data was captured using an event compression system [3]. The flowchart editor uses a subset of the standard ANSI flowchart symbols for start, process, conditionals, loops, input, output and stop. The subset consists of the symbols that would be most used by introductory programming students. Participants were required to complete a flowchart design prior to writing the code for the solution to the exercise. Participants were not permitted to make changes to their flowchart after submitting their design but were able to reference it while writing their code. *Figure 3.1* and *Figure 3.2* show screenshots of the the web application show the interface that the participants used to complete the programming exercises.

Over the course of two weeks, students scheduled one hour time slots for each of the two programming exercises to be completed under the supervision of the facilitator in a research lab using the instrumented web-based IDE. Each facilitator was required to follow the experiment design checklist to ensure consistency. While the goal was to provide exercises that were simple enough for an advanced beginner to complete within an hour, participants were not given a time limit to complete the exercise. We also permitted participants to submit incomplete solutions if they felt they could not complete the exercise.

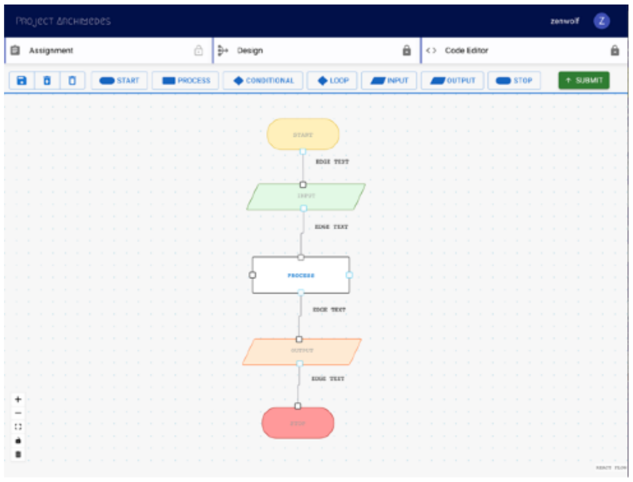
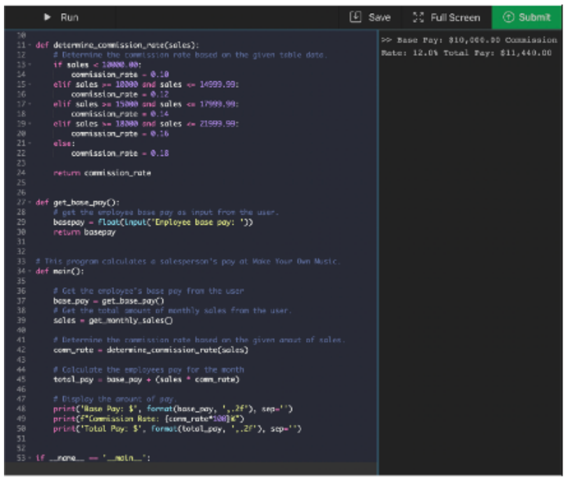


Figure 3.1 The Flowchart Design Workspace used within the Archimedes IDE



```python
def determine_commission_rate(sales):
    # Determine the commission rate based on the given table data.
    if sales < 10000.00:
        commission_rate = 0.10
    elif sales >= 10000 and sales <= 14999.99:
        commission_rate = 0.12
    elif sales >= 15000 and sales <= 17999.99:
        commission_rate = 0.14
    elif sales >= 18000 and sales <= 21999.99:
        commission_rate = 0.16
    else:
        commission_rate = 0.18

    return commission_rate


def get_base_pay():
    # get the employee base pay as input from the user.
    basepay = float(input("Employee base pay: '))
    return basepay


# This program calculates a salesperson's pay at Make Your Own Music.
def main():

    # Get the employee's base pay from the user
    base_pay = get_base_pay()
    # Get the total amount of monthly sales from the user.
    sales = get_monthly_sales()

    # Determine the commission rate based on the given amout of sales.
    comm_rate = determine_commission_rate(sales)

    # Calculate the employees pay for the month
    total_pay = base_pay + (sales * comm_rate)

    # Display the amount of pay.
    print('Base Pay: $", format(base_pay, ',.2f"), sep='')
    print(f"Commission Rate: {comm_rate*100}%")
    print('Total Pay: $', format(total_pay, ',.2f'), sep='')


if __name__ == '__main__':
```

```
>> Base Pay: $10,000.00 Commission
Rate: 12.0% Total Pay: $11,440.00
```

Figure 3.2 The Web-based IDE used to collect programming data.

3.1.1 Exercise A: Commission Rate

Exercise A was a simple exercise intended to be completed in the same one hour session of the prior programming knowledge assessment in order to orient the users the software and the process. The exercise consisted of instructions to create a program to calculate the total pay for sales staff based on the base pay, the commission rate and the amount of sales made in a month. Participants were provided with a table of sales amounts and their associated commission rate percentages, and were given examples input and output. This exercise assumes an understanding of functions, function arguments, function return values, calling functions, branching, if/elif/else statements, variables, type conversions, input and output and output formatting.

3.1.2 Exercise B: Alternating Cipher

Exercise B was a more logically complex programming problem intended to take a beginner programmer an hour to design and program. It assumes an understanding of the concepts in Exercise A in addition to working with lists, loops, and working with strings. The alternating cipher is a variation on the rail cipher and requires users to encrypt and decrypt input text by placing each character of the message on alternating lines, ignoring spaces, and then creating a single cipher text string by concatenating the two lines together. Students were given an input and output examples as well as a visualization of the encoding and decoding steps.

3.2 Participant Recruitment

Our recruitment efforts resulted in twenty-five participants ranging from freshman to senior with twelve freshmen, three sophomores, five juniors and four seniors. One participant was a graduate student. Based on the recruitment survey participants ranged from the ages of 18 to 25 with a mean age of 19.8. Four participants identified as female and 21 as male. Participant majors varied across eight disciplines. Fourteen were computer science, software engineering, or cyber security. Nine were classified as electrical or computer engineering, and the remaining three were from meteorology, chemistry and aerospace engineering, respectively. Additionally, we gathered GPA information, but note that a majority of the participants had a GPA above 3.0 on a 4.0 scale with an average of 3.7, making it difficult to make any significant claim about programming skill level based on grades. Participants were also required to have a minimum of one semester of programming experience in Python, C or C++.

Using the prior programming knowledge assessment we classified each participant with a self-estimated skill level (SE-Skill) as either Beginner, Intermediate, or Advanced in the context of introductory programming concepts. After collecting the participant data we used a case study approach to understand the initial characterization of Design Cohesion using an exploratory coding process which has led to the development of an Alignment Notation and Design Cohesion and Granularity Level as a new qualitative metrics.

3.3 Exploratory Coding Process for Classifying Design Cohesion

The classification of design cohesion requires a flowchart and its corresponding code. For Exercise A and B for each participant, the flowchart is converted into a texted-based representation of the chart. Each node in the flowchart (*Figure 3.1*) is represented by a number, its type (*i.e., input, output, process, etc*.), and the data that is contained in that node using the following structure: [number][type]:[data] as seen in *Figure 3.2* for participant P05**.** To normalize the node counts for each participant, Start and Stop nodes are not assigned a number unless they contain data other than the default 'start' and 'stop' text. If edges are used for branching or looping structures, they are included as *Branch*, and if there is a label such as True or False, the branch is assigned accordingly as *True Branch* or *False Branch*.

After the chart is converted to its text representation, the code is also converted to a text representation (*Figure 3.3*). Each line in the code is classified according to its appropriate flowchart structure, such as an input statement of input("Input monthly sales") would be given a class of [input]. An if-else structure would assign [conditional] to both the *If Branch* and the *Else Branch*. Some lines can include more than one classification, such as lines 2 and 3 in *Figure 3.4*, where input is assigned to a variable. The left side is classified as [process] and the right as [input]. Another example is when a python list comprehension is assigned to a variable. List comprehensions constitute a loop . Therefore, the left side is assigned [process] while the right is assigned [loop]. When all lines have been classified, the code is manually inspected in conjunction with the flowchart and the text-based representation of the flowchart. Next, each line is assigned a symbol using an alignment notation. The alignment notation consists of five symbols as described in *Table 3.1*. Applying the above-described classification method to Exercise A - Commission Rate for participant P05 results in 6 - *Equivalent*, 0 - *Changed*, 15 - *Added*, 0 - *MissingNodes,* and 6 - *IgnoredLines* for the example shown in *Figures 3.1 - 3.4.* Not including Comments, this results in 22 lines for the code and six nodes for the flowchart.

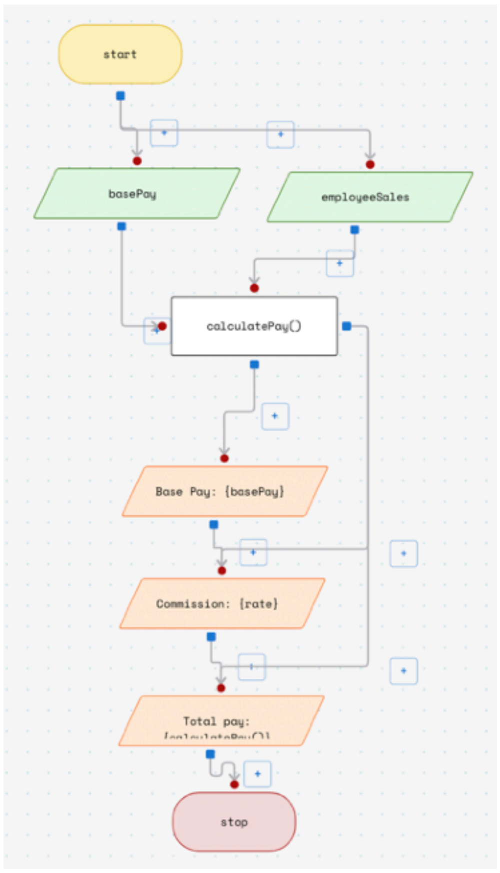
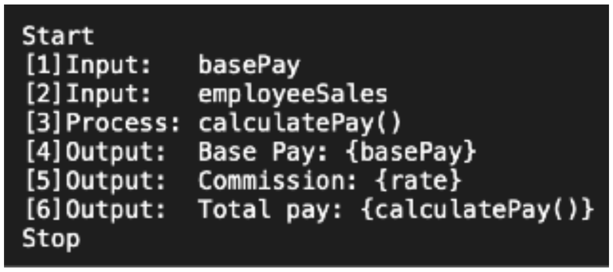
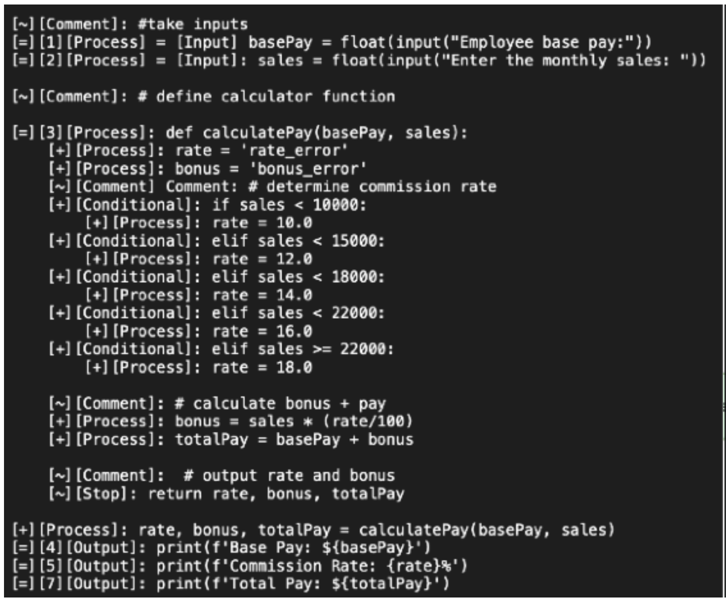


Figure 3.1: P05 Flowchart Submission Example



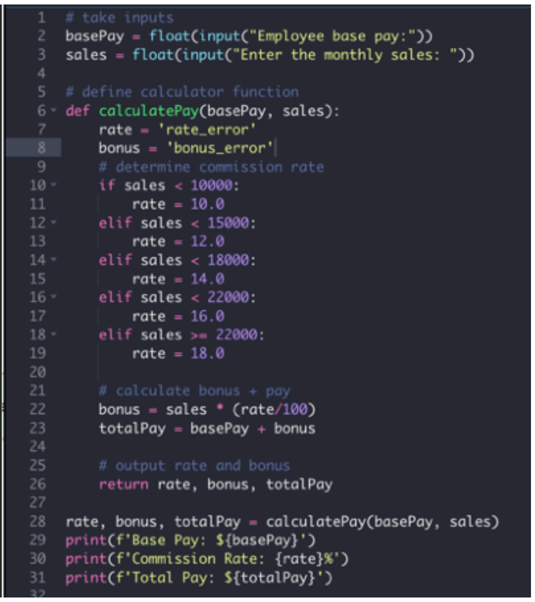
```
Start
[1]Input:   basePay
[2]Input:   employeeSales
[3]Process: calculatePay()
[4]Output:  Base Pay: {basePay}
[5]Output:  Commission: {rate}
[6]Output:  Total pay: {calculatePay()}
Stop
```

Figure 3.2: P05 Flowchart Text-based Representation



```
[~][Comment]: #take inputs
[=][1][Process] = [Input] basePay = float(input("Employee base pay:"))
[=][2][Process] = [Input]: sales = float(input("Enter the monthly sales: "))

[~][Comment]: # define calculator function

[=][3][Process]: def calculatePay(basePay, sales):
    [+][Process]: rate = 'rate_error'
    [+][Process]: bonus = 'bonus_error'
    [~][Comment] Comment: # determine commission rate
    [+][Conditional]: if sales < 10000:
        [+][Process]: rate = 10.0
    [+][Conditional]: elif sales < 15000:
        [+][Process]: rate = 12.0
    [+][Conditional]: elif sales < 18000:
        [+][Process]: rate = 14.0
    [+][Conditional]: elif sales < 22000:
        [+][Process]: rate = 16.0
    [+][Conditional]: elif sales >= 22000:
        [+][Process]: rate = 18.0

    [~][Comment]: # calculate bonus + pay
    [+][Process]: bonus = sales * (rate/100)
    [+][Process]: totalPay = basePay + bonus

    [~][Comment]:  # output rate and bonus
    [~][Stop]: return rate, bonus, totalPay

[+][Process]: rate, bonus, totalPay = calculatePay(basePay, sales)
[=][4][Output]: print(f'Base Pay: ${basePay}')
[=][5][Output]: print(f'Commission Rate: {rate}%')
[=][7][Output]: print(f'Total Pay: ${totalPay}')
```

Figure 3.3: P05 Text Representation of Code annotated using Alignment Notation



```python
# take inputs
basePay = float(input("Employee base pay:"))
sales = float(input("Enter the monthly sales: "))

# define calculator function
def calculatePay(basePay, sales):
    rate = 'rate_error'
    bonus = 'bonus_error'
    # determine commission rate
    if sales < 10000:
        rate = 10.0
    elif sales < 15000:
        rate = 12.0
    elif sales < 18000:
        rate = 14.0
    elif sales < 22000:
        rate = 16.0
    elif sales >= 22000:
        rate = 18.0

    # calculate bonus + pay
    bonus = sales * (rate/100)
    totalPay = basePay + bonus

    # output rate and bonus
    return rate, bonus, totalPay

rate, bonus, totalPay = calculatePay(basePay, sales)
print(f'Base Pay: ${basePay}')
print(f'Commission Rate: {rate}%')
print(f'Total Pay: ${totalPay}')
```

Figure 3.4: P05 Code Submission for Exercise A: Commission Rate

Name	Symbol	Description
Ignored	[~]	An *Ignored* line does not have a corresponding flowchart structure and is assigned to lines such as comments, return statements, import statements, pass, and try/except blocks. Comment lines are not included in line counts when used in subsequent calculations, but the remaining structures are.
Added	[+]	Code lines are assigned the *Added* notation if there is no node in the chart that the code line can be mapped to.
Equivalent	[=]	*Equivalent* lines are lines that map to at least one node in the flowchart text representation. In our example, we see that lines 2, 3, 6, 29, 30, and 31 are classified as equivalent and assigned a node number as they represent the corresponding nodes in the text representation. It is important to note that a single node can be equivalent to more than one line in the code, meaning that Design Cohesion cannot be defined only by directly mapping nodes to lines of code. Instead, the flowchart's level of detail, or granularity, must be considered to reduce the number of lines that are considered *Added* to the intended solution as initially represented by the flowchart.
Changed	[*]	A line is considered *Changed* if the data maps to a node in the flowchart but the flowchart construct is different such as using an output node for a process node, but the data assigns a variable.
Missing	[x]	Nodes that exist in the chart but have no code equivalent are counted as *Missing*.

Table 3.1: Alignment Notation Symbols and Descriptions

Using this alignment notation in conjunction with an expert's classification of how well the flow of the chart matches the structure of code, a Cohesion Level and Granularity Level are assigned to the participant for each exercise. Cohesion Level is a qualitative metric that takes into consideration the correctness of the intended code execution path, the nature and correctness of the data in each node of the flowchart, and how many of the nodes align with the final code solution. Granularity Level provides a classification of the level of implementation detail provided in the chart, where a high level of granularity would result in a close one-to-one mapping of flowchart nodes to code lines and a low level of granularity would represent a significant amount of encapsulation of functionality.

For the example presented in *Figures 3.1 - 3.4*, the classification process resulted in a Cohesion Level of *High* and a Granularity Level of *Low*. This classification accounts for the fact that all structures in the flowchart are implemented in the code but with a low level of implementation detail. None of the intended structures were changed, and no nodes were missing. As a result, the flow of the chart matches to a reasonable degree. However, the functionality of the *calculatePay* function is encapsulated by a single node resulting in a reduction in the level of detail. Ideally, a more accurate chart representation would use a second chart to represent the functionality and branch conditions of the calculatePay function, as high levels of encapsulation or 'black-boxing' can leave a lot of problem-solving to be completed in the coding phase where a participant may need more time to complete the project or run into unanticipated problems.

3.4 Design Cohesion and Granularity Level

After applying the alignment notation to each of the exercise samples we determined that *Design Cohesion* could be classified as *low*, *medium*, or *high*. A low level of design cohesion can indicate a low level of metacognition and ability to plan prior to implementing a programming solution. It may also represent a lack of attention to the planning phase, where a student prefers to arrive at a solution through a trial and error approach, which is referred to as a 'tinkerer' [15]. A medium level of cohesion can represent a sufficient attempt at planning before the programming phase but significant errors were made in the design phase or changes were made to the final implementation after realizing that the initial plan was not sufficient. A high level of cohesion will indicate that the student may have a high level of metacognition, and fully understood the programming problem and the programming concepts needed to implement the final solution. The data captured from the web application will be used to determine if we can automatically classify students' level of metacognition through their
level of design cohesion.

Similar to Design Cohesion *Granularity Level* can also be class as *low*, *medium* or *high.* This metric was discovered as a result of our exploratory investigation and is a subclassifications of Design Cohesion that helps us understand the level of detail contained within the flowchart. This resulted in nine potential classifications for a sample (i.e. *High Cohesion - High Granularity*, *High Cohesion - Low Granularity, etc.)* Definitions for each level of Design Cohesion and Granularity are provided in our codebook in *Table 3.2.*

In the following sections, we discuss two case studies to further demonstrate the application of the coding process. These two cases are from the same participant. The first case study looks at Exercise A, which is classified as *High Cohesion - High Granularity.* The second case presents Exercise B, classified as *High Cohesion - Low Granularity.*

3.5 Case Study: P12 - High-High vs High-Low - SE-skill Intermediate

The first cases study focuses on the data captured for Exercise A: Commission Rate from participant P12. It is classified as *High Cohesion-High Granularity*, and Exercise B is classified as High Cohesion - Low Granularity to make it clear why Granularity is a necessary feature. This participant has a Self-Estimated Programming Skill of Intermediate, a Self-Estimated Programming Experience in their primary language of C or C++ for 12 months and has a solid B GPA of 3.2 out of 4.0. In addition, they scored 14 out of 20 on the Prior Programming Knowledge Assessment (PPKA) for Python Programming.

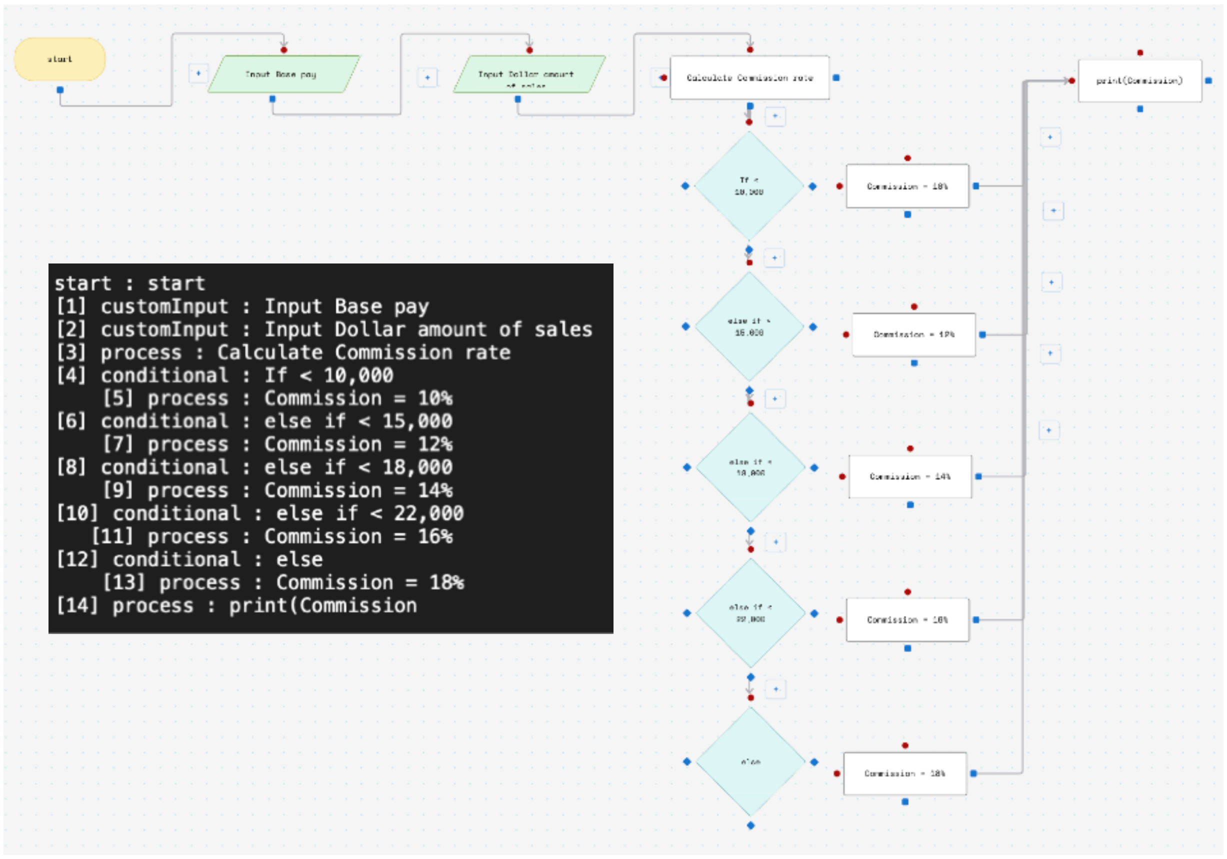
3.5.1 Case Study A: Exercise A - Commission Rate - High-High

In Exercise A, there are 14 Nodes in the flowchart (*Figure 3.5*) and 16 lines in the code solution (*Figure 3.6*). The participant took a total duration of 43 minutes (rounded) to complete the exercise, with the design phase taking 17 minutes and the code phase 26 minutes. Each chart

Design Cohesion Level	
High Cohesion	All or Most of the Flowchart Nodes map to the code, few or no programming constructs are missing or misplaced. The Node data correctly describes the resulting code. The flow chart accurately reflects the code execution order to a high degree.
Medium Cohesion	Most flowchart nodes map to the code but there are errors in the use of flowchart constructs or missing nodes. i.e. constructs in the code that are not represented in the chart. The node data mostly reflects the resulting code with some errors or ommissions. The flowchart partially relects the code execution order to some degree but there are noticable differences or errors between the code execution order and the flowcharts execution order.
Low Cohesion	Few nodes in the flowchart map to the code directly, there are many constructs in the code that are not represented in the chart or there are many errors in type of construct or in the order of the placement of the constructs in the flowchart. The data in the nodes is incorrect, vague or missing. There may be significant differences or errors between the code execution order and execution order
Granularity Level	
High	The flowchart constructs map almost one-to-one with the code. May represent over designing or explaining. Nearly all the code is represented within the flowchart.
Medium	The flowchart constructs map well to the code but do not attempt to represent every line in the code. There is an appropriate level of encapsulation of programming functionality and the design gives a reasonable level of detail.
Low	The flowchart contains a significant amount of encapsulation and black boxing of functionality and does not include a sufficient level of detail to properly document the code implementation. A significant portion of the code is represented by only a few nodes and may represent under-designing or lack of effort.

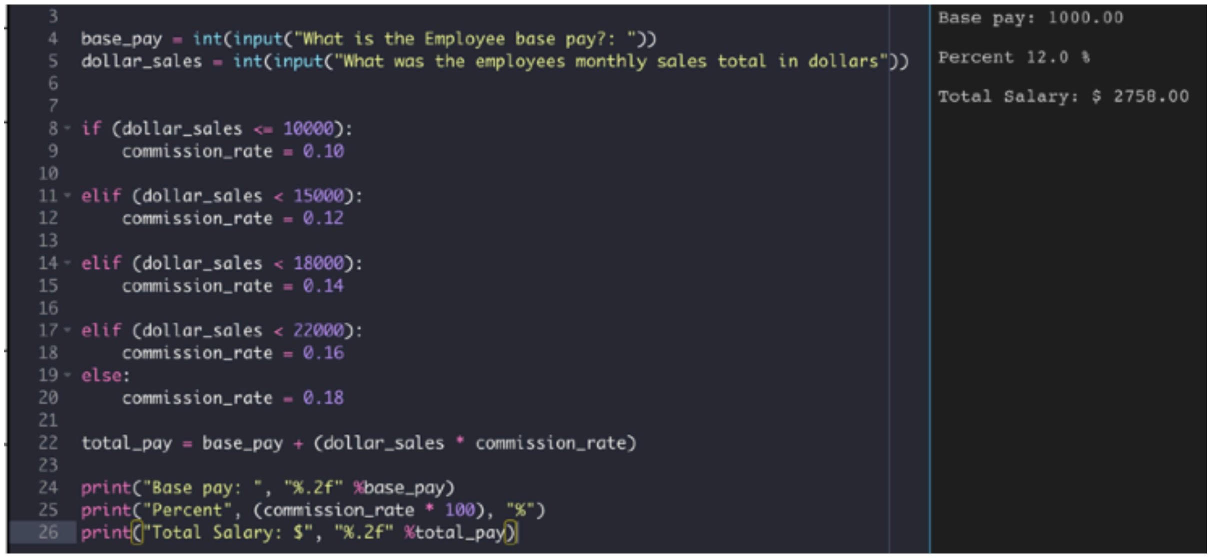
Table 3.2 Classification Criteria for Design Cohesion and Granularity Level

node contains data that is very close to the Python code used in the solution, so the chart was given a Chart Data Characterization Type of *Code* as opposed to *Pseudocode*, *Natural Language,* or *Mixed***.** With a high level of *Granularity*, we see that each line in the code maps almost one-to-one with the chart, with 12 of 16 lines marked as *Equivalent*. Lines 8 and 22 are marked as *Changed* as the conditional on line 8 was modified from "<" to "<=," and line 22 is a Process. However, the data in flowchart node three indicates a calculation of the commission rate but is calculating total pay using the commission rate as determined by the conditional. Zero lines are *Ignored*, and only two are *Added*, lines 24 and 26, for outputting the results, which are not represented in the flowchart. No nodes were counted as missing from the flowchart. The final annotated code is shown in *Figure 3.7.* As a result of all lines being accounted for by the chart and all the nodes being accounted for in the code as well as the execution flow being mostly true to the original intention, this exercise is classified as *High Cohesion - High Granularity*.



```
start : start
[1] customInput : Input Base pay
[2] customInput : Input Dollar amount of sales
[3] process : Calculate Commission rate
[4] conditional : If < 10,000
    [5] process : Commission = 10%
[6] conditional : else if < 15,000
    [7] process : Commission = 12%
[8] conditional : else if < 18,000
    [9] process : Commission = 14%
[10] conditional : else if < 22,000
   [11] process : Commission = 16%
[12] conditional : else
    [13] process : Commission = 18%
[14] process : print(Commission
```

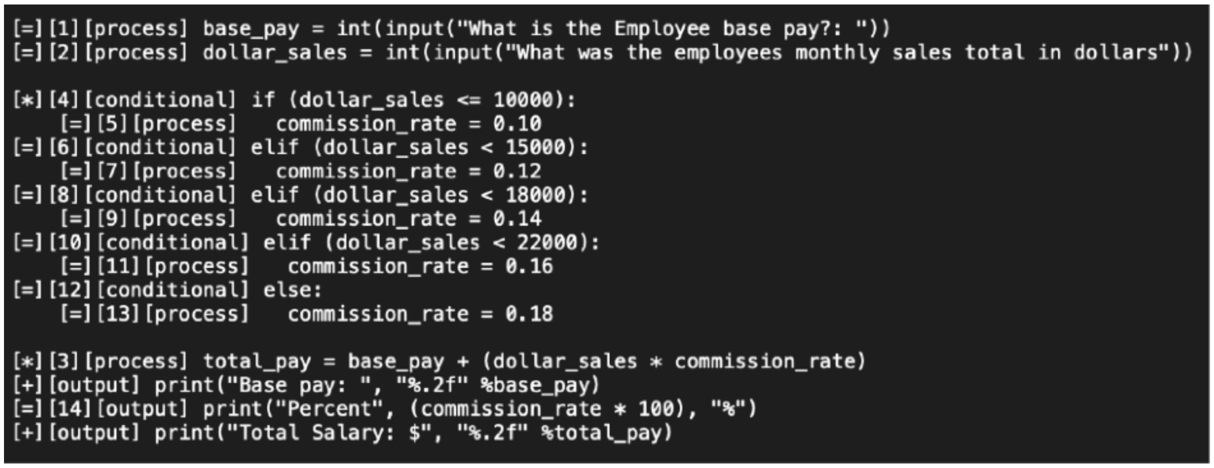
Figure 3.5: P12 - Flowchart Submission and Text Representation for Exercise A: Commission Rate



```python

base_pay = int(input("What is the Employee base pay?: "))
dollar_sales = int(input("What was the employees monthly sales total in dollars"))


if (dollar_sales <= 10000):
    commission_rate = 0.10

elif (dollar_sales < 15000):
    commission_rate = 0.12

elif (dollar_sales < 18000):
    commission_rate = 0.14

elif (dollar_sales < 22000):
    commission_rate = 0.16
else:
    commission_rate = 0.18

total_pay = base_pay + (dollar_sales * commission_rate)

print("Base pay: ", "%.2f" %base_pay)
print("Percent", (commission_rate * 100), "%")
print("Total Salary: $", "%.2f" %total_pay)
```

```
Base pay: 1000.00

Percent 12.0 %

Total Salary: $ 2758.00
```

Figure 3.6: P12 Exercise A: Commission Rate Code Solution



```
[=][1][process] base_pay = int(input("What is the Employee base pay?: "))
[=][2][process] dollar_sales = int(input("What was the employees monthly sales total in dollars"))

[*][4][conditional] if (dollar_sales <= 10000):
    [=][5][process]   commission_rate = 0.10
[=][6][conditional] elif (dollar_sales < 15000):
    [=][7][process]   commission_rate = 0.12
[=][8][conditional] elif (dollar_sales < 18000):
    [=][9][process]   commission_rate = 0.14
[=][10][conditional] elif (dollar_sales < 22000):
    [=][11][process]   commission_rate = 0.16
[=][12][conditional] else:
    [=][13][process]   commission_rate = 0.18

[*][3][process] total_pay = base_pay + (dollar_sales * commission_rate)
[+][output] print("Base pay: ", "%.2f" %base_pay)
[=][14][output] print("Percent", (commission_rate * 100), "%")
[+][output] print("Total Salary: $", "%.2f" %total_pay)
```

Figure 3.7: P12 - Text Representation of Code with Alignment Notation

3.5.2 Case Study B: Exercise B -Alternating Cipher - High-Low

In contrast to Exercise A, participant P12 seemed to alter their approach to the design phase for Exercise B and spent significantly less time on design with a duration of 3 minutes (rounded). The entire exercise duration was 32 minutes, and the code phase took 29 minutes. While the overall project and design duration was shorter, the participant took slightly longer to complete the second exercise. The short duration of the design phase is evident in the flowchart, which only contains four nodes (*Figure 3.8*) and four lines in the text representation (*Figure 3.9*), with the start node being ignored, so only three lines of code were mapped as *Equivalent.* The remainder of the code, aside from a single *Changed* and *Ignored* line, is considered additional without any node to map it to in the code. We consider this High Cohesion because all the nodes map to the code but with Low Granularity, meaning the level of detail in the flowchart is so low that it does not sufficiently document the intended design of the code. The tabulations of the features used for understanding the *Design Cohesion* and *Granularity* Classification are given in *Table 3.4* and are applied in *Figure 5.10* to the final solution shown in *Figure 3.11*.

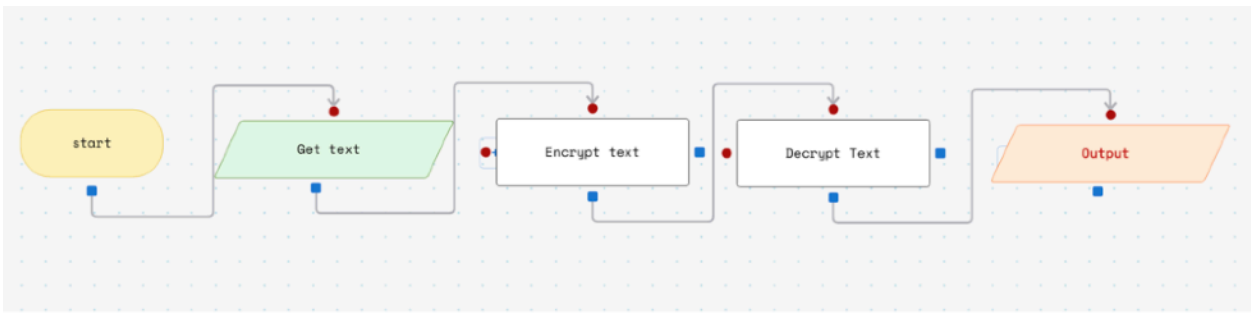
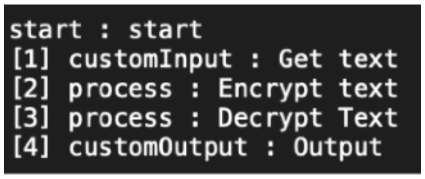
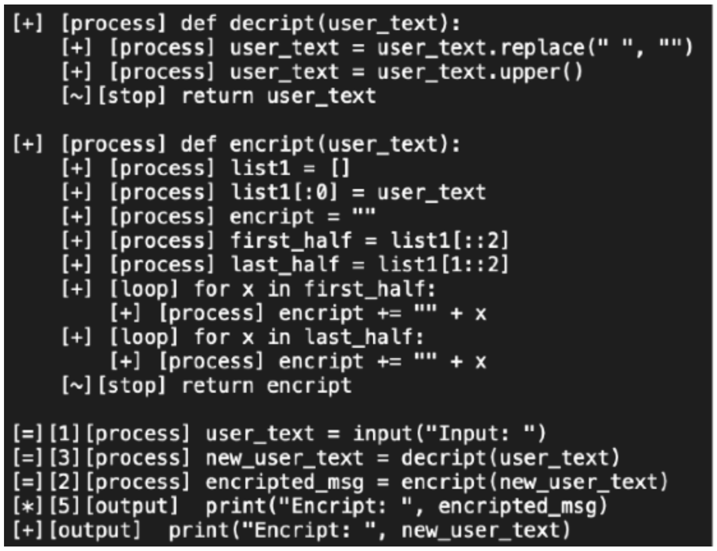


Figure 3.8: P12 Flowchart Submission for Exercise B: Alternating Cipher



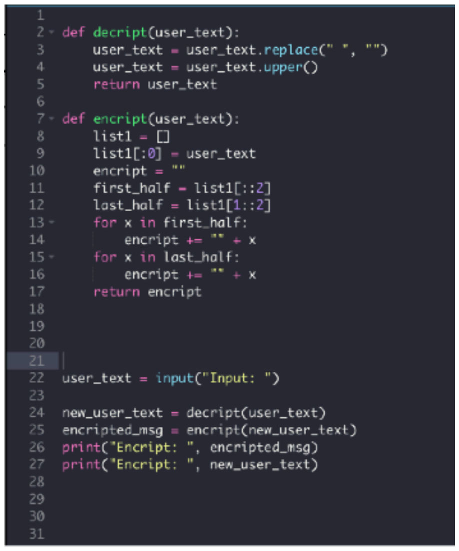
```
start : start
[1] customInput : Get text
[2] process : Encrypt text
[3] process : Decrypt Text
[4] customOutput : Output
```

Figure 3.9: P12 Exercise B Text Representation



```
[+] [process] def decript(user_text):
    [+] [process] user_text = user_text.replace(" ", "")
    [+] [process] user_text = user_text.upper()
    [~][stop] return user_text

[+] [process] def encript(user_text):
    [+] [process] list1 = []
    [+] [process] list1[:0] = user_text
    [+] [process] encript = ""
    [+] [process] first_half = list1[::2]
    [+] [process] last_half = list1[1::2]
    [+] [loop] for x in first_half:
        [+] [process] encript += "" + x
    [+] [loop] for x in last_half:
        [+] [process] encript += "" + x
    [~][stop] return encript

[=][1][process] user_text = input("Input: ")
[=][3][process] new_user_text = decript(user_text)
[=][2][process] encripted_msg = encript(new_user_text)
[*][5][output]  print("Encript: ", encripted_msg)
[+][output]  print("Encript: ", new_user_text)
```

Figure 3.10: P12 Exercise B: Text Representation of Code with Alignment Notation



```

def decript(user_text):
    user_text = user_text.replace(" ", "")
    user_text = user_text.upper()
    return user_text

def encript(user_text):
    list1 = []
    list1[:0] = user_text
    encript = ""
    first_half = list1[::2]
    last_half = list1[1::2]
    for x in first_half:
        encript += "" + x
    for x in last_half:
        encript += "" + x
    return encript




user_text = input("Input: ")

new_user_text = decript(user_text)
encripted_msg = encript(new_user_text)
print("Encript: ", encripted_msg)
print("Encript: ", new_user_text)
```

Figure 3.11: P12 Exercise B: Alternating Cipher Solution

4. Results and Discussion

In this paper we have demonstrated the application of the alignment notation and the codebook for design cohesion which defines two new metrics: Design Cohesion and Granularity Level. In doing so we have provided a new answer to our research question. We asked what metrics can be developed for assessing students' level of design cohesion based on the alignment between flowchart design and source code implementation and with this case study we have shown the development of a *Design Cohesion* classification process to manually classify the alignment between a flowchart and its associated code.

Using the newly developed metrics, we saw that one could have a High level of *Cohesion* and a Low level of *Granularity* which characterizes a high level of abstraction and points to potential under-design by the participant. In contrast, one could have High Cohesion and High *Granularity*, which could represent over-designing. In this case, each node in the chart has almost a one-to-one mapping to the code implementation. Medium *Granularity* is often more desirable than High *Granularity* in a flowchart as its effectiveness as a design and documentation tool works better at a reasonable level of abstraction than over or under-designing components. Determining the appropriate level of detail and encapsulation is often a difficult balance for Beginner and Intermediate students, with Beginners under-designing due to a possible lack of programming knowledge and Intermediate students over-designing. Advanced users are much more likely to have a Medium level of abstraction that encapsulates the finer implementation details. With this, we answer our research question in that we can create a metric to assess students' level of design cohesion as our analysis has resulted in two qualitative metrics: *Design Cohesion and Granularity Level*.

4.1 Limitations

This work has made a significant effort to collect data and develop instrumentation for establishing new metrics for IDE-based learning analytics for programming skill estimation. However, we acknowledge that there were limitations to the study. The primary limitation to this study was its size, but this was an effort to collect exploratory data to determine if using flowchart data would be useful in determining a student's level of metacognition and in the development of Design Cohesion as a component of the Model of Programming Skill Estimation. This data presents many fascinating questions and additional analysis and results for this research effort is forthcoming as we continue to explore and collect additional data. Additionally, we are working towards automation for classification of Design Cohesion and Granularity Level as it is a time consuming process. This will allow us to further validate and replicate our results as well as integrate it into our current model for providing automated feedback as we continue to improve upon and expand our methods.

We would also like to recognize that there are limitations related to the current development of the Design Cohesion metrics. Importantly, the analysis is limited because fewer students were familiar with using flowcharts as a design tool than what was assumed at the time of the study design. Previously, a flowchart design was required for all programming labs, but after COVID-19, labs were no longer conducted in person. Flowcharts were only taught to some sections of students and made optional for assignments, so many participants lacked experience with flowcharts. To mitigate this, participants were provided additional information on an as-needed basis on flowcharts, which may have biased or skewed results. At this time, it is difficult to determine if poor flowchart design was due to lack of flowchart skill, lack of programming skill, or lack of effort. In addition to the limitation related to participants' limited knowledge of flowcharts, the qualitative coding of the Design Cohesion level is further limited in that all the classification was conducted by a single person, while the person conducting the classification has significant expertise in programming, flowcharts, and design, additional raters would strengthen the validity of the classification and could be used to establish trust with the calculation of an inter-rater reliability.

4.2 Future Work

From our initial review we hypothesize that a participant with an Self-Estimated skill of Beginner will have a higher likelihood of having either Medium or Low Cohesion while a participant with a Self-Estimated Skill of Intermediate is more likely to have either medium or High Cohesion. Secondly we also hypothesize that advanced participants have a higher likelihood of having a Medium level of Design Cohesion. Our results also suggest that it is easier to draw a boundary between Low and Medium Cohesion than between Medium and High Cohesion and that it may be more beneficial to cluster the Medium and High into a single group and use *Design Cohesion* to distinguish two groups instead of three. Finally, we hypothesize that the actual skill as determined by the prior programming knowledge assessment (PPKA-Score) relates to participants' *Design Cohesion* in that if a participant has a lower PPKA-Score, they are more likely to have a Low level of *Design Cohesion* and if they have a high PPKA-Score they are more likely to have either Medium or High Cohesion. Future work and additional data will be used to to determine if these patterns are typical.

5. Conclusion

In this paper we presented our initial investigation into design cohesion as a metric for understanding introductory programming metacognition. This has lead to the development of a codebook that defines two new qualitative metrics: Design Cohesion and Granularity Level. To clarify the development of these metrics we presented our alignment notation process and used two case studies to demonstrate the reasoning behind our coding process. This has led to the discovery that high-level Design Cohesion paired with low levels of flowchart granularity demonstrates high levels of abstraction in the initial flowchart design. We postulate that this may point to under-designing by participants and/or lower levels of metacognition (either due to lack of effort or to a lack of understanding). Comparatively, having high cohesion and granularity points to over-designing by the participant and often stems from a one-to-one mapping of flowchart nodes to lines of code. Overall, our initial results point toward a logical relationship between Design Cohesion and students' level of self-estimated skill, and provide evidence that Design Cohesion will serve as a viable metric for developing our understanding of introductory students programming metacognition.

Acknowledgement
This material is based upon work supported by the National Science Foundation under Grant No. DUE- 1612132. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] P. J. Beck, M. J. Mohammadi-Aragh, C. Archibald, B. A. Jones, and A. Barton, "Real-time metacognition feedback for introductory programming using machine learning," in *2018 IEEE Frontiers in Education Conference (FIE)*, IEEE, 2018, pp. 1–5. Available: https://ieeexplore.ieee.org/abstract/document/8658973/

[2] P. Beck, M. J. Mohammadi-Aragh, and C. Archibald, "An initial exploration of machine learning techniques to classify source code comments in real-time," in *2019 ASEE Annual Conference & Exposition*, 2019.. Available: https://peer.asee.org/an-initial-exploration-of-machine-learning-techniques-to-classify-source-code-comments-in-real-time.

[3] P. J. Beck and M. J. Mohammadi-Aragh, "Archimedes: Developing a Model of Cognition and Intelligent Learning System to Support Metacognition in Novice Programmers," in *2020 IEEE Frontiers in Education Conference (FIE)*, IEEE, 2020, pp. 1–5. Available: https://ieeexplore.ieee.org/abstract/document/9274133/.

[4] P. J. Beck and M. J. Mohammadi-Aragh, "Board 421: Using a Timeline of Programming Events as a Method for Understanding the Introductory Students' Programming Process," in *2023 ASEE Annual Conference & Exposition*, 2023. Available: https://peer.asee.org/board-421-using-a-timeline-of-programming-events-as-a-method-for-understanding-the-introductory-students-programming-process

[5] P. Blikstein, M. Worsley, C. Piech, M. Sahami, S. Cooper, and D. Koller, "Programming Pluralism: Using Learning Analytics to Detect Patterns in the Learning of Computer Programming," *Journal of the Learning Sciences*, vol. 23, no. 4, pp. 561–599, Oct. 2014, doi: 10.1080/10508406.2014.954750.

[6] C. D. Hundhausen, D. M. Olivares, and A. S. Carter, "IDE-Based Learning Analytics for Computing Education: A Process Model, Critical Review, and Research Agenda," *ACM Trans. Comput. Educ.*, vol. 17, no. 3, pp. 1–26, Sep. 2017, doi: 10.1145/3105759.

[7] P. Ihantola *et al.*, "Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies," in *Proceedings of the 2015 ITiCSE on Working Group Reports*, Vilnius Lithuania: ACM, Jul. 2015, pp. 41–63. doi: 10.1145/2858796.2858798.

[8] A. Luxton-Reilly et al., "Introductory programming: a systematic literature review," in Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education, Larnaca Cyprus: ACM, Jul. 2018, pp. 55–106. doi: 10.1145/3293881.3295779.

[9] E. R. Lai, "Metacognition: A literature review," *Always learning: Pearson research report*, vol. 24, pp. 1–40, 2011.

[10] T. J. McGill and S. E. Volet, "A Conceptual Framework for Analyzing Students' Knowledge of Programming," *Journal of Research on Computing in Education*, vol. 29, no. 3, pp. 276–297, Mar. 1997, doi: 10.1080/08886504.1997.10782199.

[11] M. J. Mohammadi-Aragh, P. Beck, A. K. Barton, and B. A. Jones, "A Case Study of Writing to Learn to Program: Codebook Implementation and Analysis," in *2019 ASEE Annual Conference & Exposition*, 2019. Available: https://peer.asee.org/a-case-study-of-writing-to-learn-to-program-codebook-implementation-and-analysis

[12] S. Rum and M. Ismail, "Metacognitive awareness assessment and introductory computer

programming course achievement at university," *Int. Arab J. Inf. Technol.(IAJIT)*, vol. 13, pp. 667–675, 2016.

[13] S. N. M. Rum and M. A. Ismail, "Metocognitive support accelerates computer assisted learning for novice programmers," *Journal of Educational Technology & Society*, vol. 20, no. 3, pp. 170–181, 2017.

[14] V. Rus, M. Lintean, and R. Azevedo, "Automatic Detection of Student Mental Models during Prior Knowledge Activation in MetaTutor.," *International working group on educational data mining*, 2009, Available: https://eric.ed.gov/?id=ED539089

[15] S. Turkle and S. Papert, "Epistemological pluralism and the revaluation of the concrete," *Journal of Mathematical Behavior*, vol. 11, no. 1, pp. 3–33, 1992.

[16] J. H. Zhang, B. Meng, L.-C. Zou, Y. Zhu, and G.-J. Hwang, "Progressive flowchart development scaffolding to improve university students' computational thinking and programming self-efficacy," *Interactive Learning Environments*, vol. 31, no. 6, pp. 3792–3809, Aug. 2023, doi: 10.1080/10494820.2021.1943687.

[17] R. Smetsers-Weeda and S. Smetsers, "Problem Solving and Algorithmic Development with Flowcharts," in *Proceedings of the 12th Workshop on Primary and Secondary Computing Education*, Nijmegen Netherlands: ACM, Nov. 2017, pp. 25–34.

[18] T. Crews, "Using a flowchart simulator in a introductory programming course," *Computer Science Teaching Centre Digital Library, Western Kentucky University, USA. http://www.citidel.org/bitstream/10117/119/2/Visual.pdf*, 2001, Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=1f84050f88708298c672301822efc9e29c243c99.

[19] D. Hooshyar, R. B. Ahmad, M. H. N. M. Nasir, S. Shamshirband, and S. J. Horng, "Flowchart-based programming environments for improving comprehension and problem-solving skill of novice programmers: a survey," *IJAIP*, vol. 7, no. 1, p. 24, 2015, doi: 10.1504/IJAIP.2015.070343.

[19] N. Chapin, "Flowcharting With the ANSI Standard: A Tutorial," *ACM Comput. Surv.*, vol. 2, no. 2, pp. 119–146, Jun. 1970, doi: 10.1145/356566.356570.

[20] F. Vazquez Penaloza and C. R. Jaimez Gonzalez, "Towards a web application to create flowcharts for supporting the teaching-learning process of structured programming courses.," 2019, Available: http://ilitia.cua.uam.mx:8080/jspui/handle/123456789/297.

[21] R. R. Gajewski and E. Smyrnova-Trybulska, "Algorithms, programming, flowcharts and flowgorithm," *E-Learning and Smart Learning Environment for the Preparation of New Generation Specialists*, pp. 393–408, 2018.

[22] B. Shneiderman, R. Mayer, D. McKay, and P. Heller, "Experimental investigations of the utility of detailed flowcharts in programming," *Commun. ACM*, vol. 20, no. 6, pp. 373–381, Jun. 1977, doi: 10.1145/359605.359610.

[23] M. J. Mohammadi-Aragh, P. J. Beck, A. K. Barton, D. Reese, B. A. Jones, and M. Jankun-Kelly, "Coding the coders: A qualitative investigation of students' commenting patterns," in *2018 ASEE Annual Conference & Exposition*, 2018. Available: https://peer.asee.org/coding-the-coders-a-qualitative-investigation-of-students-commenting-patterns