Modeling cognitive processes underlying computer programming

Cătălin F. Perțicaș
Romanian Institute of
Science and Technology
perticas@rist.ro

Bipin Indurkhya
Romanian Institute of
Science and Technology
and Jagiellonian University
indurkhya@rist.ro

Abstract

We present an approach to modeling computer programming as a cognitive process. In particular, we apply Piaget's four-stage model of cognitive development to study how programming is learnt by adult programmers. For this purpose we survey software developers at different stages in their career. In order to evaluate our approach, we analyze the gathered data through formal methods. Our approach is interdisciplinary in that it incorporates philosophical, psychological, cognitive and computer science aspects.

The results from this study will be used as a starting point for investigating the role of deeper cognitive processes underlying programming, which can offer hints to design improved neural architectures inspired from biology and cognitive science. Our vision is to use such models to generate programs automatically given the intention of the user.

The preliminary goal is to set connections between empirical evidence of how programmers write code, the cognitive processes implicated in software development and the corresponding mechanisms integrated in modern neural architectures. In the future, we plan to explore the potential of such enhanced neural models to solve tasks involving generation of computer programs.

1. Introduction

In recent years, there has been much interest in automatic generation of programs (Parisotto et al. (2016), Balog et al. (2017), Ling et al. (2016), Ling et al. (2017), Yin & Neubig (2017)). These computer programs are either induced - meaning that a neural network learns to behave like the desired program, or synthesized - the neural network is trained to output a program in a language of choice. We are interested in broadening these studies by designing and experimenting with computational agents capable of learning more diverse patterns of code.

Motivated by the past successes of neural architectures inspired from the human thought process and brain structures, we begin exploring cognitive mechanisms underlying programming, which can be modeled mathematically and ultimately implemented in a neural structure yielding a computational agent capable of writing useful code.

In the research described here, we are interested in modeling the cognitive processes underlying computer programming, and then using this model to automatically generate programs given the user intentions. We take our point of departure from our earlier work on relating Piaget's interaction view of cognition with software engineering (Indurkhya (2002); Indurkhya (2003)). We incorporate more recent research on applying Piaget's theories to model programming (Corney et al. (2012); Lister (2011); Swidan & Hermans (2017); Teague & Lister (2014)). Then we take steps in the following direction:

- Research on how programming is viewed: paradigms, styles, conventions, abstractions
- Integration of research in survey design
- Collecting data for survey
- Modeling survey data for sub-groups and concepts analysis
- Investigation of extracted concepts from a cognitive perspective
- Implementation of cognitive processes inside a computational agent that generates programs

2. Philosophy of Programming

Over the years, programming has been viewed in a myriad of ways. For example, Graham (2003) argues against the traditional view that programming is a science, and relates the process of programming with the process of sketching or painting, thus linking coding or hacking to creative activities, rather than logico-deductive reasoning. (See also Hermans (2017)) Some of the pioneers of computer science, such as Edsger Dijkstra and Donald Knuth, have also subscribed to this view.

For instance, Dijkstra (1971) emphasizes the importance of good taste and style in programming by making the following analogy: teaching programming like a teacher of composition at conservatory - instead of teaching how to compose a particular symphony, help pupils find their own style. Similarly, Knuth (1968) titled his monograph, which laid the foundation of computer science, *The Art of Computer Programming.* He argued that computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty.

Indurkhya (2002) noted, *"Software is a rather unique entity. On one hand it can be considered a mathematical object — its component parts and operations of construction are rigorously defined, and the output result of a piece of software can be predicted precisely, at least in principle. On the other hand, it is also an empirical object — a piece of software executing on a machine is a physical object that can, as most of us must have experienced on many occasions, produce unexpected and unforeseen behavior. Moreover, as with any physically machinery, one can experimentally tinker with a piece of software and observe the consequences empirically."*

Such is the case with machine learning, computational physics and other computational sciences, which involve programming for the purpose of simulating theoretically defined processes. The building blocks of these fields of study are mathematical models and simulation methods. For instance, a typical neural network is equipped with both a mathematical model represented by the structure of the neurons (layering, fully-connected, shared weights, recurrent and skip connections) and their operations (weighted sums, non-linear activations); as well as with a learning/optimization method (gradient descent, nearest neighbors).

Thus, neural networks, which combine symbolic software with numerical software, are programming entities which reside as objects in two separate spaces: the mathematical world - an architectural, biologically inspired object; and the empirical, experimental world implementing the dynamics of the interaction of complex systems, which to some extent replicate the human thought processes.

3. Psychological Factors in Style Formation

By interviewing different groups of programmers and non-programmers, we seek to gather evidence to support the view that our motivations, background and everyday activities shape the type of programmer, engineer or computer scientist we can become. Moreover, the tools we use, and the people we interact with, develop and influence our vision of a skilled programmer. Later on, if our training path is successful, we get to use these skills in new creative ways. However, getting to the creative stage requires, but is not guaranteed by years of experience, mentors with insights, and a healthy learning process.

In his book, *Mastery*, Greene (2012) presents the stories of creative geniuses from a similar perspective: how their background, motivations and mentors shaped their path to mastery. The apprenticeship model developed during the Renaissance period is of key importance here. There is a nice story - Zarnescu (2007) about C. Brancusi leaving the workshop of A. Rodin, both of them being very influential sculptors of the 19th century. Brancusi stated that *"nothing grows under the shade of big trees"*. Their styles are obviously very different - as can be seen in Figure 1 - almost as if Brancusi was purposely trying to distance himself from the influence of Rodin.

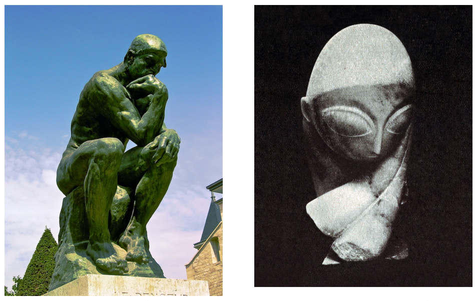


*Figure 1 – **Left**: A. Rodin's Thinker **Right**: C. Brancusi's Madame Pogany*

A first glance at the two artworks reveals that Rodin put many details in his work, while Brancusi was an essentialist. Thus, Rodin's work reveals intent, while Brancusi's work implies it and leaves some room for interpretation. We can extrapolate this simple idea to programming.

It is very common for programmers to be fond of certain programming languages because they differ in style and expressiveness. For instance, someone who wants more control of variable types and syntax checks - a 'detailist' will prefer C++ or Java, while someone who prefers simplicity - an 'essentialist' will prefer Python or Ruby. Of course, the complexity of these programming languages goes beyond this trivial example, but our purpose is to argue that programming exhibits a style component.

Identifying more specific programming styles can be done by surveying across entire programming communities. It is interesting to notice that historically programming styles translated into programming paradigms. These paradigms evolve gradually inside programming communities and can lead to the development of new programming paradigms. From this point of view, software engineering exhibits both a cultural component, as well as an innovation component.

4. Cognitive Development through Stages

We aim to analyze the evolution of a programmer from a beginner to an expert in terms of Piagetian stages of cognitive development. In particular, our goal is to tease out changes in the thought processes of learners that allow them to make quantum improvements to progress to the next level. Towards this goal, we plan to conduct an empirical study based on the surveys of non-programmers and programmers at different levels (from novice to expert).

There have been similar studies in the past for incipient stages of learning to program (novice programmers) - Lister (2011), Corney et al. (2012), Teague & Lister (2014), Swidan & Hermans (2017). However, these studies have mainly focused on the educational aspect of programming (how to teach programming); whereas we are interested in studying developmental shifts which span longer time frames, from apprenticeship to mastery.

These developmental shifts reveal changes in how certain already existing cognitive processes are used in coding by experienced programmers, while they are not by novice programmers. Some of these processes can actually be learned quickly and prove very useful, but because they are not required until a later stage, their development is paused. However, the ones involving more effort to grasp, but are required for reaching a more immediate stage, are more heavily invested into.

4.1. Stage One: Simple Reflexes

Understanding the building blocks. It has been argued that mathematics is grounded in human activities (*Mathematics, Form and Function*, Mac Lane (1986); *Where mathematics come from*, Lakoff & Nunez (2000); *The Number Sense*, Dehaene (1996)). Piaget has also argued that mathematical concepts arise gradually from sensorimotor actions in *Biology and Knowledge*, Piaget (1971). Based on this, we hypothesize that how we write programs is highly influenced by our interests and by our simple interactions with the computer.

Human Activity	Related Math Idea	Math Technique	Programming Construct
Collecting	Object Collection	Set, class, multi-set, list, family	Array, list, objects and instances
Connecting	Cause and Effect	Ordered pair, relation, function, operation	Dictionary, Graph algorithms
Endless repetition	Infinity, Recursion	Recursive set, infinite set	For and while loops, recursion

Figure 2 – An extension to connections drawn from Mathematics, Form and Function

To test this hypothesis, we conduct short interviews of some recently 'self-made' programmers, who had different jobs before but decided to learn programming without any formal computer science education, and eventually managed to get jobs as software developers. We expect our analysis to reveal that what they were doing before drove their current interests in programming. Moreover, their concept of programming is influenced by the tasks they were doing before via analogies.

Figure 10 shows some information extracted from the interviews conducted. Most investigated cases in the 'self-made' sub-group revealed that programming is viewed as a tool - the means to achieve a goal. Side interests related to programming are either derived from their background or technology trends common in their work group. Their other side interests influence their interactions and role within their work-group, as well as their developmental vision - whether they are theoretically inclined, heading towards research directions, or practically inclined, thus preferring the engineering side of programming.

4.2. Stage Two: Pre-Operational

Functional thinking. According to Piaget, at this stage children do not yet understand concrete logic and cannot mentally manipulate information. Children's interest in playing and pretending also takes place in this stage. However, the child still has trouble seeing things from different points of view. The children's play is mainly categorized by symbolic play and manipulating symbols.

To study the cognitive processes underlying programming corresponding to this second stage, we choose to observe the behavior of a few summer interns at a research institute. Generally, internships are used by modern day companies to create proof of concepts for some ideas for which internal staff cannot be allocated. Interns are viewed as helpers, who are motivated by learning practical skills in a working environment, in a similar fashion to the apprenticeship model detailed by Greene (2012).

Given their motivation for learning and a lack of practical experience, we could consider their activities as playing in a work environment, which would correspond to Piaget's pre-operational stage. The interns typically do not see their work from a business perspective, which is consistent with Piaget's observation that children do not see things from a different perspective at this stage.

Figure 12 displays some of the attributes and tasks performed by the surveyed interns. We found that they managed to come up with good research ideas based on the topic they were provided with. Their imagination was quite rich and given proper guidance, they showed the ability to translate some of their ideas into actual software. However, they did not have so much success with the more complex topics they wanted to explore, either because they were not able to articulate their vision well, or because their technical expertise was not yet good enough.

At this stage, they still need someone to help select the most promising ideas out of the technically feasible ones. Another important aspect was their steadiness in solving the more challenging tasks. This seemed to be associated with both internal factors - how happy they were with their research topic and their work flows; as well as with external topics - the amount of encouragement they received for following their ideas.

4.3. Stage Three: Operational

During this stage, Piaget noted that a child's way of thinking starts to be more adult-like. Problems are solved in a more logical fashion. However, abstract thinking is not yet developed so children can only solve problems that apply to concrete events or objects. Nonetheless they can generalize by making inferences from observations.

As a setting for the third stage of development, we chose to observe the environment of high-school programming competitions. The reason behind this choice is that good competitors are people who already know how to program, who have developed their problem-solving skills and implementation abilities, and who are able to generalize concepts across various types of problems.

At this stage, programmers can swiftly manipulate common patterns or templates used in programming competition problems. Moreover, they can reliably estimate the necessary time to implement a well-defined idea, as well as to precisely put their solution in practice. The book *Psychology of Coding Competitions*, Francu (1997) is particularly addressed to such high-school students. It suggests various strategies for training and problem solving.

The take-away message is that problem solving skills, which incorporate the ability to transform conceptual ideas into concrete algorithms, do not guarantee the success in competitions without a well-defined strategy belonging to the domains of psychology, decision making and time management. These concepts are mostly reflected in the thought processes of adult minds.

Towards the end of this stage, we can already observe the need for strategical thinking, which according to Piaget, is a cognitive process that is predominantly observed in the next developmental stage. Going back to the high-school programming competitions, we find that mentors play a key role in conveying strategical planning and decision making. Figure 13 shows focus areas and other features of various mentors we surveyed.

4.4. Stage Four: Formal Operational

Piaget's theory states that abstract thinking, meta-cognition and problem solving are developed during this stage. In programming, these thinking patterns translate into a deeper understanding of programming concepts and how they relate to each other.

In the field of programming, the fourth stage of development is best represented by senior developers, who typically have more than 5 years of experience (industry standard). This stage coincides with the crystallization of the programmer's application domain or specialization.

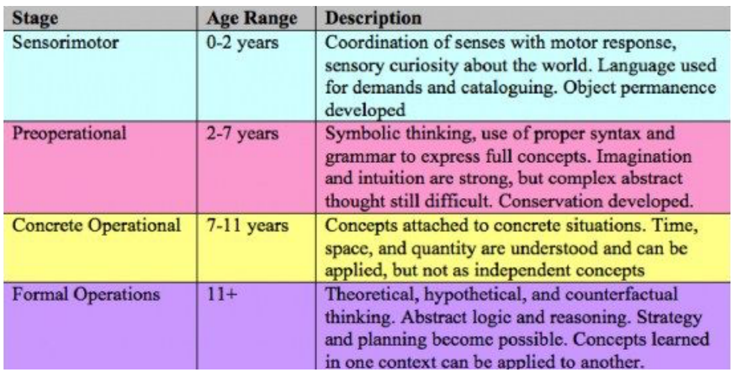
Senior programmers show a high degree of technical expertise in their specialization (in-depth knowledge). They have a broad view of other specializations (wide knowledge). And they are able to independently create fairly complex software or test research ideas end-to-end. Figure 11 displays information on surveyed senior programmers.

5. Formalizing Piagetian Attributes for Concept Analysis

The surveys we have conducted so far are insightful from a psychological point of view: we can observe trends in the developmental process of a programmer and how their background and motivations influence their learning curve and application domain.

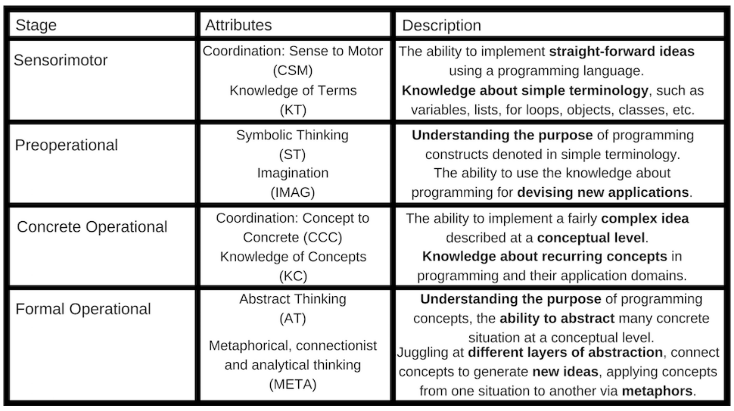
However, different participant groups have been analyzed from different angles. At the same time, the data gathered is expressed in natural language. This makes it hard to make rigorous empirical observations that would generalize to new participants. For this reason, we started working on a set of principles to make such studies more generalizable and easier to interpret.

Our first principle is to convert a sample of Piaget's developmental stages into the corresponding stages for learning to write programs. The second principle is to have a list of skills/thinking patterns specific to each developmental stage - Figures 3 and 4. These skills are converted into attributes, which can be subjectively quantified based on surveys - Table 5.



Stage	Age Range	Description
Sensorimotor	0-2 years	Coordination of senses with motor response, sensory curiosity about the world. Language used for demands and cataloguing. Object permanence developed
Preoperational	2-7 years	Symbolic thinking, use of proper syntax and grammar to express full concepts. Imagination and intuition are strong, but complex abstract thought still difficult. Conservation developed.
Concrete Operational	7-11 years	Concepts attached to concrete situations. Time, space, and quantity are understood and can be applied, but not as independent concepts
Formal Operations	11+	Theoretical, hypothetical, and counterfactual thinking. Abstract logic and reasoning. Strategy and planning become possible. Concepts learned in one context can be applied to another.

Figure 3 – Summary of Piaget's development stages from The Psychology Notes Headquarter - image source.



Stage	Attributes	Description
Sensorimotor	Coordination: Sense to Motor (CSM) Knowledge of Terms (KT)	The ability to implement **straight-forward ideas** using a programming language. **Knowledge about simple terminology**, such as variables, lists, for loops, objects, classes, etc.
Preoperational	Symbolic Thinking (ST) Imagination (IMAG)	**Understanding the purpose** of programming constructs denoted in simple terminology. The ability to use the knowledge about programming for **devising new applications**.
Concrete Operational	Coordination: Concept to Concrete (CCC) Knowledge of Concepts (KC)	The ability to implement a fairly **complex idea** described at a **conceptual level**. **Knowledge about recurring concepts** in programming and their application domains.
Formal Operational	Abstract Thinking (AT) Metaphorical, connectionist and analytical thinking (META)	**Understanding the purpose** of programming concepts, the **ability to abstract** many concrete situation at a conceptual level. Juggling at **different layers of abstraction**, connect concepts to generate **new ideas**, applying concepts from one situation to another via **metaphors**.

Figure 4 – Piaget's stages applied to programmers (our model) and the description of stage-emergent attributes/thinking patterns.

Id	CSM	KT	ST	IMAG	CCC	KC	AT	META	YoE
A.1	1.0	1.0	1.0	0.75	1.0	0.5	0.5	0.5	6
A.2	1.0	1.0	1.0	1.0	1.0	0.75	0.75	0.5	4
A.3	0.75	0.75	0.75	1.0	0.25	0.5	0.25	0.0	3
B.1	0.75	0.5	0.5	0.5	0.25	0.25	0.0	0.0	3
B.2	1.0	0.75	0.75	0.5	0.75	0.25	0.25	0.0	2
B.3	0.75	1.0	1.0	0.75	0.5	0.5	0.75	0.25	1
B.4	1.0	0.75	0.75	0.5	0.75	0.5	0.0	0.25	3
C.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.75	12
C.2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.75	10
C.3	1.0	1.0	1.0	1.0	1.0	1.0	0.75	1.0	14
C.4	1.0	1.0	1.0	1.0	1.0	0.75	0.75	0.75	8

*Figure 5 – **Id**: Programmer Ids, **A**'s represent interns at a programming/research job, with formal education in computer science, but little practical experience. **B**'s represent programmers with formal education in a different field, but practical software skills. **C**'s are senior programmers with formal education and practical skills. Next 8 columns are subjective quantifications of different skill levels according to our scheme for Piaget's stages applied to programming. These are drawn from the conducted surveys. **YoE** represents the number of years they have been exposed to programming.*

The third principle in analyzing this data is to use a mathematical tool for drawing conclusions from the data. Because our goal is to extract thinking patterns of programmers, and find out how these apply to various sub-groups of programmers, we model the data through a concept lattice taken from the theory of *Formal Concept Analysis* Ganter & Wille (1999) - Figure 6.

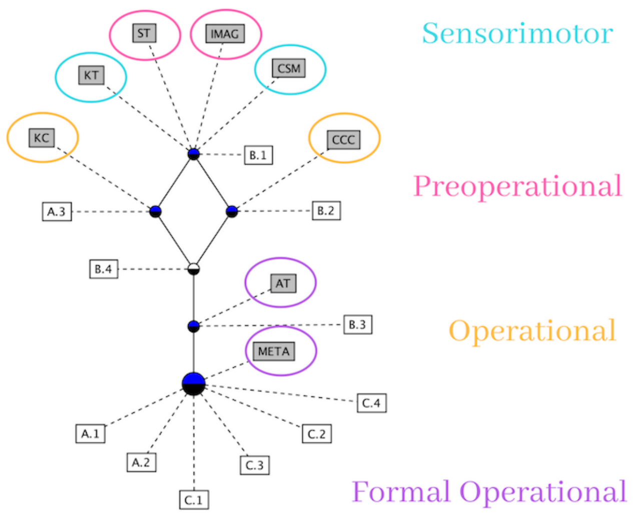


*Figure 6 – **The Concept Lattice** derived from Table 5 by thresholding subject-attribute connections if skill level is above 0.5. The lattice can be interpreted the following way: a subject (A.1,..., B.1,..., C.4) has an attribute (KC, KT,..., META) if there is an ascending path from the subject to the attribute. The unlabeled nodes represent concepts that exist in the formal context. Concepts are formally defined as maximal subsets of subjects sharing a maximal subset of attributes.*

Our findings show that there is a strong agreement between Piaget's theory, as we modeled it for programmers (attributes and skill levels), and their corresponding years of experience. We now get to the fourth and final principle of exploring this type of empirical surveys, which is to analyze this data through emergent concepts (6 in this case) - Table 7.

Formal Concept Name	**Subjects**	**Description**
Sensorimotor + Preoperational	100%	All have coordination, knowledge of terminology, symbolic thinking & imagination.
Almost Operational Type A	A.3 & below	Knowledge of concepts, but low conceptual to concrete - **theoretically inclined**.
Almost Operational Type B	B.2 & below	Conceptual to concrete, but low knowledge of concepts - **practically inclined**.
Operational	B.4 & below	Abstract and creative thinking not yet developed for programming.
Postoperational	B.3 & below	B.3 has 1 year of programming experience and a strong mathematical background => **abstract thinking** developed.
Formal Operational	A.1, A.2 & C's	All these programmers have at least 4 years of experience.

*Figure 7 – **Interpretation of Formal Concepts** The concepts in the formal context from Figure 6 exhibit a close correspondence to Piaget's stages for programmers - 3 concepts represent exact stages (Preoperational, Operational and Formal Operational), while the remaining 3 represent intermediate stages of development.*

Even though the number of participants and attributes are limited in our study, this principled method can be applied to more complex datasets for the discovery of human concepts in a formally defined context. The subjectivity in the evaluation of programming skills can be overcome via triangulation or objective evaluations, such as problem solving tests.

6. Cognitive Processes underlying Programming

By going into more depth, we can investigate the role of our cognitive processes in solving specific types of problems involving the design of algorithms, their implementation and general problem-solving in programming environments. Our brains have developed a number of complex cognitive mechanisms and systems for finding solutions to problems, some of these are inherent, natural properties of the brain, such as the ability to perceive and to attend to certain stimuli, while others are emergent and require developmental transitions, such as those exemplified within Piaget's theory.

Out of these, we investigate the role of perception and attention. These seem to play an important role, not just in programming, but in daily human activities, as well as in computational agents designed to mimic the human mind. Variants of these mechanisms have been formalized and integrated into applications of neural networks.

6.1. Perception

For instance, perception at the level of neural networks has produced different kinds of architectures. These range from the traditional multi-layer perceptron (MLP), which do not assume any correlation between input units, to convolutional neural networks (CNN) mimicking the visual processing system by assuming spatial correlations; and recurrent neural networks (RNN) which exploit temporal correlation of input units.

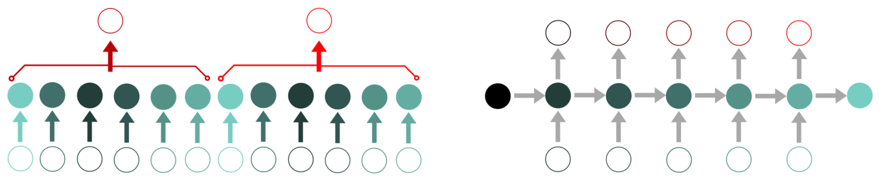


*Figure 8 – **Left**: CNN Perception. **Right**: RNN Perception.*

It it interesting to notice how non-programmers view code as hieroglyphics. Even programmers find it hard to decipher code with obfuscated or badly named variable names. Although variable and function names do not influence the computational aspect of source code, they have a high impact on our understanding and perception of the code because they serve as anchors or starting points for creating internal representations of code. Only internally well represented code can allow for useful manipulations of source code.

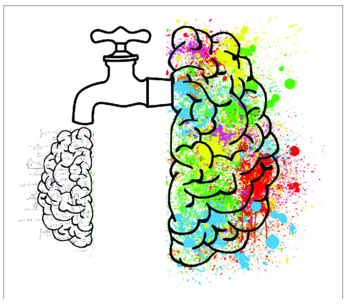


Figure 9 – A. Huxley's Cerebral Reducing Valve - image source

Internal representations of the problem setting are key ingredients in general human problem solving. These internal representations are obtained through filtering and abstraction of innate perceptions. These reduction mechanisms are developed for maximizing our adaptive fit and to help with efficiency and clarity. However, because of their top-down modulation - sensory system is subordinated to conceptual thinking system - solutions that do not fall in the common known patterns can be found in the background of our minds, but they are eliminated. Huxley (1954) offers a nice perspective of this idea in his book *Doors to Perception*, arguing that our minds tend to reduce, rather than produce - Figure 9.

6.2. Attention

Top-down modulation in cognitive systems is usually referred to as attention, which is the next process we are interested to explore. According to Jones (1890), attention is *"taking possession of the mind, in a clear and vivid form"*. However, more modern psychologists - Cherry (2018), claim that attention is both a highlighter, as well as the withdrawal from some things in order to deal more effectively with others.

Thus, attention is limited, selective and it is a core part of any cognitive system. This seems natural given the limited number of sensors present in cognitive systems. Attention is required in order for perception to be meaningful; attention guides perception. The interaction between these 2 systems generates the ability to select limited, but useful information from an unbounded noisy environment. While perception represents bottom-up information processing, attention is top-down modulation based on expectation.

Programming requires the ability to focus. Whether the object of focus is a line of code, an entire procedure or a project, being able to write or change code functionality would not be possible without attending to key components and places. For instance, changing one line of code might affect the desired effect of other lines of code, not necessarily in the proximity of the changed line. It is important to pay attention to the parts of code that are conceptually related to the changed area of code because in most cases it is impossible to have a perfect view of the whole.

Attention plays an important role in recurrent neural networks used for machine translation (NMT). For years it seemed hard to model long-term dependencies in recurrent neural models. The Long Short-Term Memory (LSTM) model was specifically designed to deal with this issue and saw great success. However, the issue still persisted for sequence-to-sequence problems. The attentional mechanisms described by Bahdanau et al. (2015) and Luong et al. (2015) significantly improved the state of the art in automatic translation of a sentence from one language into another.

Intuitively, the improvement comes from the learnt ability to attend to certain key phrases instead of the whole sentence. Because the neural architectures are of the encoder-decoder type, the information extracted by using the whole sentence maps to a conceptual representation of what the sentence means as a whole. Details such as the gender of a noun or verb tense are lost in this representation. On the other hand, attention allows to process a sentence one phrase at a time, thus ensuring that low-level information is correctly integrated in the translation.

Similar attention mechanisms have been tested on visual problems, such as the one presented by Xu et al. (2016) for generating image captions, or the one by Mnih et al. (2014) for localization and detection of handwritten digits. Neural networks implementing attention exhibit yet another advantage, which is to provide humans the ability to see what they see. For instance, attention weights will highlight the area in the image used for producing a given result.

7. Conclusions

We started this study by analyzing how programming is regarded and by establishing its similarities to arts and crafts. This revealed the fact that programming has many sides to it, which are shaped by the psychological factors influencing the person who creates programs. The development of programming skills and tendencies was then modeled using Piaget's cognitive theory. We applied this model to data extracted from interviews of various groups of programmers and realized the importance of quantifiable attributes when testing the reliability of our model. For this reason, we gathered a sample of numerical attributes denoting a variety of skill levels in programming for different subjects. Cognitive patterns corresponding to Piaget's theory were then discovered using formal context analysis. Finally, we mentioned some cognitive processes involved in programming, which could be included in our model to improve its generality. Our plan is to gather more data and use our concept discovery model to find recurring cognitive patterns involved in programming, which can be integrated in an automated code generator agent, the same way attention and perception were used to improve performance of neural networks.

8. Acknowledgement

This work was supported by the European Regional Development Fund and the Romanian Government through the Competitiveness Operational Programme 2014-2020, project ID P_37_679, MySMIS code 103319, contract no. 157/16.12.2016.

References

Bahdanau, D., Cho, K., & Bengio, Y. (2015). Neural Machine Translation by jointly Learning to Align and Translate. In *3rd International Conference on Learning Representations (ICLR).*

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., & Tarlow, D. (2017). DeepCoder: Learning to Write Programs. In *5th International Conference on Learning Representations (ICLR).*

Cherry, K. (2018). *How Psychologists Define Attention.* Retrieved from https://www.verywellmind.com/what-is-attention-2795009

Corney, M., Teague, D., Ahadi, A., & Lister, R. (2012). Some Empirical Results for Neo-Piagetian Reasoning in Novice Programmers and the Relationship to Code Explanation Questions.

Dehaene, S. (1996). *The Number Sense*. Oxford University Press.

Dijkstra, E. (1971). *A Short Introduction to the Art of Programming*. Technische Hogeschool Eindhoven.

Francu, C. (1997). *Psihologia Concursurilor de Informatica*. L&S Infomat.

Ganter, B., & Wille, R. (1999). *Formal Concept Analysis: Mathematical Foundations*. Springer.

Graham, P. (2003). *Hackers and Painters.* Retrieved from http://www.paulgraham.com/hp.html

Greene, R. (2012). *Mastery*. Viking Press.

Hermans, F. (2017). Code as Art - Art as Code: On the Use of Poetry and Paintings in Programming Education.

Huxley, A. (1954). *The Doors of Perception*. Chatto & Windus.

Indurkhya, B. (2002). On Philosophical Foundation of Lyee: Interaction Theories and Lyee. *New Trends in Software Methodologies, Tools and Techniques, IOS Press: Amsterdam, in H. Fujita and P. Johannesson (eds.)*, 45–51.

Indurkhya, B. (2003). Some Philosophical Observations on the Nature of Software and their Implications for Requirement Engineering. *New Trends in Software Methodologies, Tools and Techniques, IOS Press: Amsterdam, in H. Fujita and P. Johannesson (eds.)*, 29–38.

Jones, W. (1890). *Principles of Psychology*. Henry Holt and Company.

Knuth, D. (1968). *The Art of Computer Progrmming*. Addison-Wesley.

Lakoff, G., & Nunez, R. (2000). *Where Mathematics Comes From*. Basic Books.

Ling, W., Grefenstette, E., Hermann, K., Kocisky, T., Senior, A., Wang, F., & Blunsom, P. (2016). Latent Predictor Networks for Code Generation. *arXiv:1603.06744v2*.

Ling, W., Yogatama, D., Dyer, C., & Blunsom, P. (2017). Program Induction by Rationale Generation: Learning to Solve and Explain Algebraic Word Problems. *arXiv:1705.04146v3*.

Lister, R. (2011). Concrete and Other Neo-Piagetian Forms in the Novice Programmer.

Luong, M., Pham, H., & Manning, C. (2015). Effective approaches to Attention-based Neural Machine Translation. In *Conference on Empirical Methods in Natural Language Processing EMNLP*.

Mac Lane, S. (1986). *Mathematics, Form and Function*. Oxford University Press.

Mnih, V., Heess, N., Graves, A., & Kavukcuoglu, K. (2014). Recurrent Models of Visual Attention.

Parisotto, E., Mohamed, A., Singh, R., Li, L., Zhou, D., & Kohli, P. (2016). Neuro-Symbolic Program Synthesis. *arXiv:1611.01855v1*.

Piaget, J. (1971). *Biology and Knowledge*. University of Chicago Press.

Swidan, A., & Hermans, F. (2017). Programming Education to Preschoolers: Reflections and Observations from a Field Study.

Teague, D., & Lister, R. (2014). Blinded by their Plight: Tracing and the Preoperational Programmer.

Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhutdinov, R., … Bengio, Y. (2016). Show, Attend and Tell: Neural Image Caption Generation with Visual Attention.

Yin, P., & Neubig, G. (2017). A Syntactic Neural Model for General-Purpose Code Generation. *arXiv:1704.01696v1*.

Zarnescu, C. (2007). *Codul Operei lui Brancusi*. Dacia.

Id	Current Job	Job Type	Prog Lang	Interests
C.1	Machine Learning	Research	Python	Cognitive Sciences and AI
C.2	iOS Developer	Industry	Swift	Hacking and Cryptocurrency
C.3	Machine Learning	Research	JavaScript	Neural Networks
C.4	WordPress Developer	Freelancer	C#	Robotics and Arduino

Figure 11 – Table illustrating different types of senior developers.

Id	Background	Current Job	Side Interests	Insights
B.1	design & crafts music	front end & web design	personal growth & spirituality digital art & synthesized music	Helping the team to learn new technologies & looking for new ways to improve built products. Programming viewed as a tool mostly.
B.2	social sciences	full stack web developer	politics & law	Always in a search for new challenging tasks because it stimulates learning. Programming is great because it is a highly-demanded job.
B.3	mathematics	machine learning programmer	neuroscience & artificial intelligence	Reading & thinking about new research directions. Programming is great because you can test ideas quickly.
B.4	journalism	quality assurance tester	machine learning & automatic testing	Reading about the potential of AI to disrupt & innovate technologies. Programming is interesting because of the social effects it has.

Figure 10 – Table with illustrating data from 'self-made' programmers, with no formal education, such as a university degree in computer science.

Id	Predisposition	Background	Suggested Research Topic	Articulation	Effectiveness	Steadiness
A.1	Technical	Web Scraping	Automatic Web Scraping from DOM features	high	medium-high	high
A.2	Tech. Creative	Debugging	Automatic Bug Finding from ASTs	high	high	medium-high
A.3	Creative	Games	Multi-Agent Learning Behavior in Simple Games	medium	medium-high	medium

Figure 12 – Table illustrating data from interns in their 2nd year of computer science program.

Id	Focus Area	Background	Explored Topics	Structure
H.1	Learnability	High-school Teacher	Basic programming constructs and techniques, fundamental algorithms and the thinking behind them	Synthesized
H.2	Strategic Thinking and Problem Solving	Doctorand and Ex-Contestant	Advanced problem solving, decision making and strategy in programming competitions and psychological preparation	Structured
H.3	Creative Problem Solving, Simplification, Analogies and Approximations	Current Contestant	Hacking, unconventional programming, style of thinking	Unstructured

Figure 13 – Table illustrating different types of computer science mentors.