Programming Style: Influences, Factors, and Elements

Anthony Cox
Centre for Psychology and Computing
Dartmouth, Nova Scotia, Canada
amcox@cenpsycom.org

Maryanne Fisher
Saint Mary's University
Halifax, Nova Scotia, Canada
mlfisher@smu.ca

Abstract

As a consequence of various situational and personal factors, programmers use a variety of styles when performing software development and maintenance tasks. In this paper, we develop a contextual framework that links situational, task-oriented, and individual factors to a set of traits. These traits are expressed as behavioural modifiers, and thus, influence one's performance of tasks such as computer programming, as well as influencing the skills upon which programming is based (e.g., problem solving, planning). To demonstrate the use of the framework, we examine how risk-tolerance is influenced by contextual factors and show how this trait influences programming style. We also document some preliminary components of programming style.

1 Introduction

Individual differences are known to be present in computing and have been identified in specific activities such as using a compiler [13] or more generally, in software development [28] or when using computers to perform "management information systems (MIS) tasks" [6]. Since individual differences have been documented with respect to human-computer interaction, it is likely that educational, cultural, social, and other factors will influence activities such as computer programming. As a consequence of these factors, each programmer will exhibit her or his own *programming style*. That is, programmers will differ in how they generate source code and in, for example, their work habits, the order in which they implement a system's components and their use of programming tools. In this paper, we explore some of the key factors that influence programming style, and develop a comprehensive framework that explains individual differences when programming.

To unify previously identified program comprehension approaches, von Mayrhauser and Vans [29] developed an integrated comprehension model. In their model, programmers opportunistically shift between top-down and bottom-up approaches. Similarly, Kato and Takeuchi [14] have found that the cognitive skill of way-finding is an opportunistic process and that individuals with a better sense of direction made more effective use of their strategies and switched appropriately. Thus, we expect that programmers, though tending to have a preference towards specific, favoured programming strategies, are likely to switch strategies when it becomes beneficial, possible, or necessary. Programming style must accordingly be viewed as a set of preferences, and not as immutable characteristics.

Green, Bellamy and Parker [10] observe that programmers do not generate code linearly and will jump forwards and backwards. Mosemann and Wiedenbeck [20] document that during program comprehension, programmers do not examine a file sequentially. This movement of a programmer within a source file is referred to as *software navigation*. Fisher, Cox, and Zhao [8] examined the use of spatial cognition during program comprehension to explore programmer's movements within source code, but not the reasons for these movements. While software development is the underlying root cause for software navigation, individual differences should greatly impact one's strategy when performing software development. For example, it has been shown that one's experience is a factor when selecting a program comprehension strategy [21]. We suggest that navigation is a key component of programming style and use it to form the basis for our definition of *programming strategy*. We define the term programming strategy with respect to the implementation order of a program's sub-elements and consider a strategy as an order of development for, and hence navigation through, a system's components.

2 Motivation

Our initial intent was to experimentally explore and model the concept of programming style and to identify the factors that influence programmer behaviour. The first factor that we considered was risk tolerance as it is known to affect decision making [30] and consequently, programming style. That is, stylistic differences between program-

978-0-7695-3529-6/09 $25.00 © 2009 IEEE
DOI 10.1109/ACHI.2009.48

mers can be viewed as the result of programmers making different choices when faced with decisions.

Our experiment contained a programming risks survey, the NEO FFI personality inventory [4], and a programming style survey. We also had participants perform a software development activity in a highly instrumented environment. We hypothesised that a programmer's risk tolerance would affect activities such as the amount of code they worked on and the frequency with which they used the compiler or tested the program. After preparing the stimuli, we recruited 8 participants to pilot test the experiment. In previous work, we had found that pilot testing was crucial in avoiding the effects of unexpected confounds. Participants were given an unstructured interview after performing the experiment.

When we examined the data and interviews, we realised that we had made many errors. We had not separated the elements of programming style from the traits of the programmer. For example, we had considered the preference for abstraction as an element of programming style, when it is actually a more general personality trait that affects programming style. We also discovered that there were many unconsidered situational factors that affect programming style. For example, the previous programming task that participants had performed, such as working on an assignment, or performing thesis research, affected how they responded in the experiment. Thus, we realised that programming is highly sensitive to situational influences. We were aware that risk tolerance is domain-specific [30], but had not fully considered other contextual factors.

Consequently, we became aware that it is impossible to accurately build a general model of programming style. The situational factors that impact programming style are highly influential, and thus any model is tied to these factors. This realisation lead us to develop a framework in which specific models of programmer behaviour can be described.

3 A Model-Independent Framework

There are three types of factors that influence programming style: the actual task to be performed, the situation in which the task is performed, and the person who is performing the task. Figure 1 details how these three factor sets interact to form a *programming context*. It should be noted that the task and the situation will vary with respect to the perceptions of the individual. For example, task difficulty is a factor that varies according to the individual's experience and ability to apply that experience. It can therefore be said that a context is specifically a product of the observer.

The *situation* is a complex set of factors that represents all the characteristics surrounding where, how, when and with who the task must be performed. The situation encompasses issues such as the time available, the organizational culture, and other environmental constraints. The *task* encompasses all the characteristics displayed by a specific task, such as the required functionality, programming constraints (e.g., language, development environment), and any runtime execution constraints. It is possible to view the task as another element of the situation. However, due to the highly technical nature of programming, and the focus of the framework, we consider the task to be an independent group of factors encompassing all technical elements. Therefore, while factors, such as the choice of programming language, could also be considered as a situational factor, we suggest that predominantly technical constraints are better viewed as part of the actual task. Finally, the *individual* represents the programmer performing the task within the situation. The internal component of the programmer describes all the unchangeable factors that are due to biology, such as genetics, sex, and cognitive capacity. The external component represents the programmer's education, training, and experiences. That is, external influences are events that have shaped the programmer's skills.

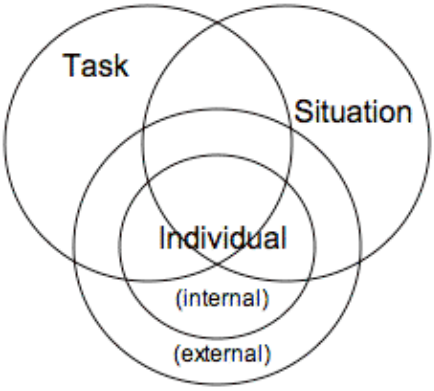

Figure 1. A Programming Context

The next element of the framework are the individual's *traits*. By definition, a trait is an element of one's personality that is not directly expressed, but is manifested through its influence on the expression of actual behaviours. For example, patience is a trait since it cannot be performed, but it can influence the performance of behaviours.

As a starting point, we suggest that the following traits will influence the task of computer programming: conscientiousness, agreeableness, creativity, risk tolerance, abstraction preference, choice of granularity, concurrency, and flexibility. In Section 4, these traits are examined in detail. We note that these traits may not be exclusive and that this set may be extended as additional research is performed.

DeHaemer [6] lists the factors that he believes influences the performance of MIS tasks using the categories: psychological, demographic, communication skills, motor skills, and motivation. There is considerable, but not total over-

lap between DeHaemer's and our factors. In general, we believe that some of his factors, such as manual dexterity (within the motor skills category), may have a small but insignificant impact on programming style. His factors are similar, in that they also show overlap, and vary from highly specific (e.g., tolerance for ambiguity) to highly general (e.g., intelligence, cognitive style).

A trait that is independent of all other traits will be referred to as a *primary trait*. If we apply the *swiss army knife model* of cognition [3], where cognitive skills are built upon a small set of highly versatile abilities, the primary traits are these abilities, or the "blades" in the knife. Secondary traits are those that are influenced by other traits. For example, one's experience influences the assessment of task difficulty and permits the formation of judgements regarding risks when performing specific development activities. Tolerance of risk (i.e., a potentially primary trait) influences the likelihood of performing activities concurrently, since one tends to avoid concurrency during risky activities. Thus, preference towards using concurrency is a secondary trait as it is influenced by risk tolerance.

We have developed a framework in which models must be rebuilt every time the situation or task changes. The framework is sensitive to changes in any influencing factor and thus links models of programmer behaviour to a specific context. The key observation that we make from this contextual dependency is that any model developed using the framework is highly specific and can not be generalised for other contexts (i.e., changes in the selection of programmer, task, or situation). We use the term *model* to describe specific, context dependent behaviour and consider our framework to be model-independent.

The reason for the highly specific perspective is that research has shown that some traits are not stable for a specific individual, and change with respect to situational factors. DeHaemer [6] summarises research which shows that the task and "context" (i.e., situation) impact the skill of decision making. As all programmers must make decisions when programming, it follows that the task and situation must be taken into account when building a framework for examining programming style.

In the framework, traits can be positioned on the context diagram to indicate whether they are influenced by individual, situational, and task dependent factors. As well, individual variation and perceptions are likely to cause the traits to move, depending on whether the individual is sensitive to various situational factors. Not only will the traits move, but due to situational influences and individual variation, the strength of the trait and its ability to influence expressed behaviour will also change. Thus, for each context, a specific *trait pattern* exists. As these traits influence programming behaviour, the trait pattern provides a means for describing the factors that influence programming style.

As well as traits, all individuals, as a consequence of their innate cognitive capabilities, education, and experiences, have a *skill web*. This web is an interconnected set of *basic* and *compound* skills. The basic skills are those that are not based on any other skills. Examples of basic skills are the spatial cognition concepts of object and location memory, or the ability to recognize a single written character. Compound skills are formed using basic skills, and often other compound skills. For example, mathematics, problem solving, and decision making, are compound skills. Computer programming is also a compound skill that is based on other compound skills, such as planning and problem solving.

When we perform a specific skill, the performance of that skill will be affected by the context in which it is performed. For example, our ability to perform complex cognitive tasks such as programming is hindered when we are tired, unmotivated, depressed, hurried, or performing a novel and unfamiliar task. As traits are defined as behavioural modifiers, a contextual trait pattern provides a mechanism by which changes in skill application and performance can be linked to contextual factors.

4 Traits

The expression of our skills is influenced by our traits and which in turn are influenced by the context in which they exist. Changing any contextual factor can potentially change the trait pattern, and consequently, the individual's expression of skills used when programming. We now examine some specific traits that influence programming style.

4.1 Risk Tolerance

Smith [26] defined risk for civil engineering projects and since software development is a form of construction, Smith's definition is applicable. In his perspective, risk is a decision based on a range of possible outcomes, with attached probabilities. Where there are no known probabilities, uncertainty, not risk, is considered to exist. Thus, risk exists in situations in which one must choose from a variety of options, of varying costs, and for which some of the options lead to more desired outcomes that occur with some element of unpredictability.

Risk tolerance, and its opposite, risk aversion, is not a simple trait for which one has a preferred tendency. Wang [30] demonstrates that an individual's perceptions can influence their risk tolerance and that risk tolerance is situational. Consequently, one's risk tolerance when programming is contextual in nature and is not a more general trait.

However, though the context is critical for determining an individual's tolerance of specific risks, it has been well established among social psychologists that men are more

risk tolerant, and take more risks, than women (see Daly and Wilson [5] for a comprehensive review). Culture also affects our risk aversion, as Carment [1] has found that Indians tend to be less risk tolerant than Canadians. Furthermore, he demonstrates that there is a relationship between risk tolerance and the one's performance when applying one's skills in an industrial training institute. This evidence suggests that activities such as computer programming will be affected by one's assessment of risk in a specific context.

Knowing that risk tolerance is dependent upon factors such as sex, culture, and context, we see that this trait should be located in Figure 1, on the line between individual internal factors (e.g., sex) and external factors (e.g., cultural influences), in the zone that is the intersection of task, individual, and situation. Furthermore, the strength of the trait, and thus its likelihood of affecting skill performance, is also affected by these same factors. When a programmer's risk tolerance is assessed for a specific context, it can then used to predict how the trait affects performance. However, as there does not exist a validated context-specific measure of risk tolerance, we are currently limited to only generalised predictions on how risk will affect programming style.

In general, when developing code, the more code one writes, the greater the risk of an error being made. Thus we expect women, who prefer lower-risk strategies, will tend to write shorter code fragments and compile more frequently than will men in order to minimise the likelihood of making errors. As well, since compilers only identify syntax errors, women should execute and unit test the code more frequently to detect other coding errors. Consequently, women should tend towards using a bottom-up development strategy to permit more frequent testing of source code.

Bottom-up program development and comprehension are low risk strategies. One works from a known position and builds using only what one knows is correct. Top-down strategies are much riskier as they can not be confirmed as correct until the supporting lower-level code is available. Thus, we suggest that women, documented as more risk averse, will prefer to use bottom-up comprehension and development strategies while men are more risk prone and are more likely to use top-down strategies.

These predictions have been verified experimentally. It has been documented that women tend to program using a bottom-up approach while men tend towards using a top-down approach [28]. Thus, it is likely that this known difference is a result of differences in risk tolerance. Similarly, there is evidence that women exhibit a similar preference for using a bottom-up program comprehension strategy [8].

4.2 Abstraction Preference

Almost all activities can be viewed as having an element of abstraction. Research has shown that one's preference for using abstraction is also influenced by a variety of variables. It has been found that, for example, when performing navigation, women tend to use less abstract, more concrete representations of the environment, such as landmarks [24].

Postma *et al.* [22] suggest that route-based navigation can be performed at both an abstract or a concrete level, but that the use of cognitive maps is predominately performed at an abstract level. If women tend towards the less abstract, then they should tend to use a route-based navigation strategy. Lawton [17] confirms this suggestion and found that women were more likely than men to use route-based strategies whereas men were more likely than women to use map-based navigation strategies. Conversely, men tend to use more abstract representations such as cardinal directions (e.g., north, south) [23]. When the level of abstraction is considered, it can be seen that bottom-up programming and landmark identification both use a low level of abstraction. Bottom up programming is focused on an application's lowest and least abstract functional units. Similarly, top-down programming and cardinal direction tend towards a higher level of abstraction. It is likely, but currently unexplored, that women will also have a better knowledge of *beacons* as they are the equivalent of landmarks within source code.

Independent of whether top-down or bottom up development is performed, programmers may view their activities as either concrete or abstract. An abstract view will be motivated by the availability of high level abstractions (e.g., an architecture), and will thus permit the programmer to view the software with respect to these abstractions. With a concrete focus, a programmer will tend towards the development of specific functionality. It might be suggested that for an abstract-oriented programmer, the design comes first and is implemented, and for a concrete-oriented programmer, the functionality comes first and the design evolves as the functionality becomes more complex.

A clear indication of a programmer's abstraction preference is their understanding and knowledge of the software's design. One can not view the system from an abstract level, without there being an appropriate set of abstractions in place. We suggest that the design provides these abstractions. However, it is possible that a programmer may use alternative abstractions such as an object hierarchy.

4.3 Personality Factors

One widely accepted and validated model of personality [12] is the NEO FFI [4]. The model categorises personality with respect to the five traits of neuroticism, extraversion, openness, agreeableness, and conscientiousness. The factors of agreeableness and conscientiousness are likely to have significant impact on computer programming.

Agreeableness affects the ability of a person to perform "skeptical and critical thinking" in that one who is agreeable

will be more prone to following the path suggested by others [4]. Agreeableness defines one's likelihood of performing critical thinking. The facet of trust that is associated with agreeableness is considered as a measure of skepticism and indicates one's ability to think independently. As independent and critical thinking are key elements of problem solving when programming, it is likely that agreeableness will have an impact on programming tasks.

Conscientiousness measures a variety of factors such as competence, order, dutifulness, striving for achievement, self-discipline, and deliberation. Conscientiousness is also related to personal motivation to learn and to perform at a higher level [2]. For example, an individual that scores highly with respect to the facet of deliberation will tend to "think carefully before acting" [4]. However, it remains unclear whether conscientiousness is always beneficial for task performance. Given that it reflects characteristics of dependendability (e.g., order, deliberation dutifulness), it might actually impede critical and independent thought [18]. It should be noted that conscientiousness is a very abstract trait and is certainly influenced by other traits such as risk tolerance and abstraction preference.

Scores on NEO FFI are affected by a variety of factors. For example, openness is related to the amount of education that one has obtained. Other demographic variables, such as age and sex are also known to correlate to specific factors. Thus, as both internal and external individual factors can influence one's expression or personality traits, it is likely that other contextual factors are also relevant.

DeHaemer [6] suggests that tolerance for ambiguity is also a factor that influences programming style. A programmer who avoids ambiguity will think more concretely, and will avoid abstraction until all details are resolved. With respect to personality, ambiguity is an element of neuroticism. Programmers that are neurotic will avoid ambiguity and prefer clarity, while programmers that display less neuroticism will be more tolerant of ambiguity. As top-down programming requires one to add detail during implementation, top-down approaches can be associated with ambiguity. Bottom-up programming is less ambiguous as programmers combine clearly defined functional components.

It is known that problem solving can take many forms and that one can prefer an algorithmic or a creative approach to solving problems [9]. Thus creativity, will have some impact on programming style. Creativity is considered to be related to extraversion, and other personality factors, and is believed to occur when one is less inhibited [16].

It is difficult to accurately measure one's potential to be creative, so most researchers use composite measures [15]. One can demonstrate creativity with respect to problem solving, artistic endeavours, or one's approach to self entertainment. It is also known that creative accomplishments are also inversely related to conscientiousness and inversely correlated with agreeableness [15]. Thus, creativity is contextual in nature, and can vary with respect to the task being performed [19].

4.4 Concurrency

The number of activities that a programmer can perform simultaneously is expressed as *concurrency.* Some programmers avoid concurrent development and work on a single task, while others work on several tasks at once. Cognitive science research reveals there are significant individual differences in preference and ability to perform concurrent tasks, and that practice on the separate tasks increases the likelihood of success when multi-tasking [25].

One factor that affects concurrency is experience. A novice is likely to use a non-concurrent approach to programming, as the task demands their full attention. However, even experienced programmers are likely to avoid concurrency when working on the most difficult algorithms and components of an application. Thus, concurrency should correlate with a programmer's experience, and with the difficulty of the implementation tasks that are being attempted.

Concurrency is related to a programmer's *cognitive load*. When cognitive load is minimal, a programmer can attempt several tasks and devote part of their capacity to each task. As the difficulty of a task increases the task requires more cognitive effort, has a higher cognitive load, and thus decreases the available capacity for other tasks [27].

While concurrency may be considered with respect to the number of instances of a single task that a programmer attempts (e.g., implementing a method) it can also be exhibited in the variety of tasks that a programmer attempts. For example, a highly concurrent programmer may simultaneously be implementing new functionality while repairing incorrect functionality (i.e., "bug" repair).

4.5 Granularity of Skill Application

When coding, a programmer can work on a small, very specific element of functionality or may alternatively work on a larger body of functionality. That is, a programmer may choose to implement a part of a method, the entire method, or an entire class. As with other factors, granularity is usually selected in concert with other factors. For example, larger bodies of code are more likely to contain errors, and are thus riskier to produce. Moreover, granularity often relates to abstraction, such that for a smaller "grain" size a task is more concrete and for a larger "grain" size a task is more complex [11].

An indicator of granularity is the frequency of compiler use. Programmers who compile frequently are using a very low level of granularity, and are doing only a minimal

amount of work before checking their syntax using the compiler. Programmers who tend to compile less frequently are implementing more source code in each compilation cycle and are working at a higher level of granularity.

5 Meta-Factors

We refer to the factors that influence the application of the framework as *meta-factors*. These elements are not part of a specific model of programming style, but affect the consistency, validity, and robustness of a generated model.

Although we suggest that programmers are opportunistic when selecting and applying a programming strategy, they may not always exhibit flexibility. Less confident or novice programmers are likely to be more repetitive and less flexible in their choice of strategy. Programmers can only select from a range or programming strategies when they are familiar with many strategies and understand each strategy's strengths and limitations. Consistency is therefore highly influenced by experience and education. With regard to programming, someone that displays a rigid or consistent programming style will not be able to identify opportunities where alternative choices will be more effective. Consistency can be viewed as the variation that one exhibits from a model developed using the framework, and is thus a meta-factor regarding the application of the framework.

Psychologists have created many models of motivation that integrate individual preference, affect, beliefs, and social norms as determinants. Recently, motivation has been examined with respect to goal selection and setting behaviour [7]. We use the term in a similar manner to describe the process, and the associated motivation, that underlies programming. That is, programmers can use a very goal-driven approach, where they focus on an end result and strive to reach that result, or they can use an activity driven approach, where they focus on an activity (e.g., coding, debugging, testing). Furthermore, motivation is susceptible to risk tolerance. For example, a risk tolerant individual may feel motivated when presented with a high-risk challenge while a risk averse individual may not. As a result of this relationship between motivation and risk tolerance, any model of programming style should take programmer motivation into consideration.

Learning is a feedback loop that changes the context as an individual changes. Thus we say that learning is not part of the framework, but explains why changes will occur within it. Learning, like consistency, is a meta-factor that influences how the framework is applied. For the sake of brevity, we do not identify a specific learning style (e.g., reflective learning) and leave this issue to future research.

6 Programming Style

We now examine various concepts that form the foundation of one's programming style. These characteristics are separate from the traits that influence programming style, as reviewed in Section 4. Here we instead focus on the actual elements of programming style.

6.1 Breadth

When implementing a system, it is possible to apply a breadth-first, or a depth-first approach. One can implement a specific functionality in detail (i.e., depth-first) or one can slowly build up the application by implementing a portion of each needed functionality by building all functionalities, and then combining them (i.e., breadth-first). It is expected that the hierarchical structure that is necessary to identify breath and depth-first approaches is provided by software's control flow, and in particular, the function call graph.

This factor is not to be confused with concurrency. One can work on several elements of a single path into a software system (i.e., depth-first and concurrent) or one can work on a single element of the path at a time (i.e., depth-first and non-concurrent). Conversely, one can apply a breadth-first approach and work on a single, or multiple, potentially unrelated program components at the same time.

With respect to software engineering methodology, breadth-first development is a form of iterative development where many features are implemented in part and improved on each iteration. Depth-first development is a form of incremental development, where functionality is added in small increments, one feature at a time.

6.2 Direction

Programmers can chose to implement an application by starting with the smallest, most specific elements first, or they can work on the most general elements. Working on specific elements can be viewed as a bottom-up approach, with respect to a directional structure, such as the aforementioned function call-graph. An adding of detail is therefore a top-down approach, where a programmer starts with the earliest called, but most general component, and then begins to add detail. A bottom-up approach causes a programmer to develop a working sub-component and then join the sub-components to form higher-level abstract component.

One is not always free to select an implementation direction. Languages such as C require that a `main` function be implemented to permit successful compilation. However, it is possible to build a minimal path to `main` and then carefully implement this path in a specific direction. As with all factors, direction is therefore a tendency, and must be applied with respect to imposed constraints.

6.3 Perspective

When viewing a system, programmers may use a domain-oriented perspective or a programming perspective. That is, the software may be considered from the perspective of the end-users, or of the developers. One can employ a view that sees a financial application with respect to financial concepts, or one can view the same system as a collection of algorithms, and data structures.

With respect to classical program comprehension, a domain-oriented perspective is equivalent to top-down comprehension where one works from the application domain towards the code. Bottom-up program comprehension is equivalent to a programming perspective where one sees the code and identifies domain-oriented abstractions within it.

Perspective is not the same as direction. One can work down, from the `main` function, adding an `initialize` and then an `openfiles` function, without ever viewing the files as representing application domain artefacts. While top-down development is more suited to an application-oriented perspective, the two factors may be applied distinctly, and are thus independent.

In current environments, the choice of perspective can have significant impact. When an API (Application Programming Interface) is provided, an application-oriented developer may try to apply the API to the application, while a practitioner with a programmer perspective is more likely to modify the application to suit the API. Thus, APIs invite, and usually reward the use of the programmer perspective as they are designed by, and for, programmers.

6.4 Individual Preference

We have defined programming strategy with respect to the implementation order of an application's software elements. In simplest terms, a strategy can be defined using a single criterion as shown in the following examples:

1. Complexity driven (e.g., easiest parts first).
2. Functionality driven (e.g., interface first).
3. Execution driven (e.g., matching of execution order).
4. Usage driven (e.g., most used first).
5. Familiarity driven (e.g., most familiar parts first).
6. Re-use driven (e.g., copy and paste some existing code).

These strategies focus on properties of the software (e.g., complexity), the programmer (e.g., familiarity), or the development context (e.g., re-use).

6.5 Summary

Programmer's work habits reflect their programming activities. That is, work habits are direct representations of traits such as concurrency and granularity. A programmer will work on a preferred amount of code, for a preferred length of time, at a preferred speed. While performing this work, a programmer may perform more than one task in a multi-tasking fashion. Thus, our work habits are directly influenced by our traits.

Programmers are not expected to consistently apply any of the identified factors, and will naturally, due to their personality and experiences (i.e., their trait pattern), work within their comfort zone for each one. As many factors are best applied in specific combinations we suggest that programmers will thus form composite strategies that exploit these combinations. The traditional approaches of top down and bottom up program development are examples of such combinations. Both approaches are naturally associated with the direction of the same name. However, bottom-up development is more than simply joining together code elements to build up functionality.

The trait of abstraction preference has an impact on the programming style elements of direction, and perspective. Top-down development is perhaps most effective when done using an abstract view based on the application domain. Using an application based perspective, it is easy to develop an architecture or design based on key application concepts. Then, one can begin the process of translating those concepts into source code. Conversely, bottom-up development tends to require a more concrete perspective based on the programming domain. When one is working closely with source code, building a program from its components, the design will evolve as one assembles those components in a bottom-up manner.

While one can associate abstraction with direction in that a concrete programmer is more likely to use a bottom-up approach, the two concepts are fully not related. A programmer may use a top-down approach, to build a path to specific functionality. We expect that abstraction is strongly linked to both direction and breadth. Programmers that avoid abstraction will likely work in a bottom up manner using a depth-first approach. We believe that these preferences have previously prevented practitioners and educators from separating the concepts of abstraction, direction, and breadth and suggest that this framework provides a new view on programming style.

7 Conclusion

In this paper, we have demonstrated the need to create a framework that allows a high-degree of flexibility in the weight given to a particular situation or task, with respect to how the individual will perform. We have argued that there are numerous individual factors that researchers must consider when examining programming style, and do not believe it is possible to create an accurate and general model of programmer behaviour that is independent of the situation and task. Instead, we encourage researchers to focus on the

individual within the context of the task and situation. Thus, our framework represents an important step in helping researchers and educators to better understand programmers and how they produce source code. Although this work is preliminary in nature and needs to be tested using a series of empirical studies, our framework provides a means for researchers to integrate the results of such studies. Thus, by combining several contemporary research findings, we offer researchers a new perspective on the behaviour of computer programmers.

References

[1] D. Carment. Risk-taking under conditions of chance and skill in India. *Journal of Cross-Cultural Psychology*, 5(1):23–35, 1974.

[2] J. Colquitt and M. Simmering. Conscientiousness, goal orientation, and motivation to learn during the learning process: A longitudinal study. *Journal of Applied Psychology*, 83:654–665, 1998.

[3] L. Cosmides and J. Tooby. Cognitive adaptations for social exchange. In *The Adapted Mind*, chapter 3, pages 163–228. Oxford University Press, Oxford, UK, 1992.

[4] P. Costa and R. McCrae. *NEO PI-R Professional Manual*. Psychological Assessment Resources, Lutz, FL, 1992.

[5] M. Daly and M. Wilson. Risk-taking, intrasexual competition, and homicide. In *Nebraska Symposium Series, No. 47: Motivation*, pages 1–36, 2001.

[6] M. DeHaemer. Vulcans, Klingons, and Humans: The relevance of individual differences for information systems interfaces. In *Computer Personnel Research Conference*, pages 156–163, 1991.

[7] C. Dweck and J. Heckhausen. *Motivation and self-regulation across the life span*. Cambridge University Press, Cambridge, UK, 1998.

[8] M. Fisher, A. Cox, and L. Zhao. Using sex differences to link spatial cognition and program comprehension. In *International Conference on Software Maintenance*, pages 289–298, 2006.

[9] A. Gallagher and R. De Lisi. Gender differences in scholastic aptitude test and mathematics problem solving among high-ability students. *Journal of Educational Psychology*, 86:204–211, 1994.

[10] T. Green, R. Bellamy, and J. Parker. Parsing and gnisrap: A model of device use. In *Empirical Studies of Programmers: 2^{nd} Workshop*, pages 132–146, 1987.

[11] J. Hobbs. Granularity. In *International Joint Conference on Artifical Intelligence*, pages 432–435, 1985.

[12] R. Holden and G. Fekken. The neo five-factor inventory in a canadian context: psychometric properties for a sample of university women. *Personality and Individual Differences*, 17:441–444, 1994.

[13] M. Jadud. A first look at novice compilation behaviour using BlueJ. *Computer Science Education*, 15(1):25–40, 2005.

[14] Y. Kato and Y. Takeuchi. Individual differences in wayfinding strategies. *Journal of Environmental Psychology*, 23:171–188, 2003.

[15] L. Kin, L. Walker, and S. Broyles. Creativity and the five-factor model. *Journal of Research in Personality*, 30:189–203, 1996.

[16] V. Kumar, D. Kemmler, and E. Holman. The creativity styles questionnaire revised. *Creativity Research Journal*, 10, 1997.

[17] C. Lawton. Gender differences in way-finding strategies: Relationship to spatial ability and spatial anxiety. *Sex Roles*, 30(11/12):765–779, 1994.

[18] J. Lepine, J. Colquitt, and A. Erez. Adaptability to changing task contexts: Effects of general cognitive ability, conscientiousness, and openness to experience. *Personnel Psychology*, 53:563–593, 2000.

[19] T. Lubart. Creativity across cultures. In R. Sternberg, editor, *Handbook of Creativity*, pages 339–350. Cambridge University Press, Cambridge, UK, 1999.

[20] R. Mosemann and S. Wiedenbeck. Navigation and comprehension of programs by novice programmers. In *International Workshop on Program Comprehension*, pages 79–88, 2001.

[21] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[22] A. Postma, G. Jager, R. Kessels, H. Koppeschaar, and J. van Honk. Sex differences for selective forms of spatial memory. *Brain and Cognition*, 54:24–34, 2003.

[23] D. Saucier, M. Bowman, and L. Elias. Sex differences in the effect of articulatory or spatial dual-task interference during navigation. *Brain and Cognition*, 53:346–350, 2003.

[24] D. Saucier, S. Green, J. Leason, A. MacFadden, S. Bell, and L. Elias. Are sex differences in navigation caused by sexually dimorphic strategies or by differences in the ability to use the strategies? *Behavioral Neuroscience*, 116:403–410, 2002.

[25] E. Schumacher, T. Seymour, J. Glass, D. Fencsik, E. Lauber, D. Kieras, and D. Meyer. Virtually perfect time sharing in dual-task performance: uncorking the central cognitive bottleneck. *Psychological Science*, 12:101–108, 2001.

[26] N. Smith. *Engineering Project Management*. Blackwell Science Ltd., Osney Mead, Oxford, UK, 2002.

[27] J. Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive Science: A multidisciplinary journal*, 12:257–285, 1988.

[28] S. Turkle and S. Papert. Epistemological pluralism: Styles and voices within the computer culture. *Signs*, 16:128–157, 1990.

[29] A. von Mayrhauser and A. M. Vans. Comprehension processes during large scale software maintenance. In *International Conference on Software Engineering*, pages 39–48, 1994.

[30] X. Wang. Self-framing of risky choice. *Journal of Behavioral Decision Making*, 17(1):1–16, 2004.