FOCUS DRIVEN DEVELOPMENT: THE "COULD" AND "SHOULD" OF SOFTWARE DESIGN

Robert Green
Department of Computer Science
Bowling Green State University
Bowling Green, OH 43403
greenr@bgsu.edu

ABSTRACT

It is not uncommon for software developers to work on significant software products that can, for the most part, be thought of as "spaghetti code" - code that appears to be haphazardly designed with little thought to structure and technique. Unsurprisingly, it is never the intention of a developer to create spaghetti code. In the beginning of a project, high quality code is commonly developed with great success. Yet, as the project continues, the developer begins to lose focus on important aspects of the software (like design and readability) and begins making poor choices under the pressure of time, money, etc. This paper explores this idea of "focus" in the development process and how a strong focus on the essentials - Location, Function, Naming, Communication, and Refactoring - can lead to better software design.

INTRODUCTION

In an article in the Harvard Business review, Greg McKeown speaks about the failure of successful people in becoming very successful due to the Clarity Paradox, or the "Undisciplined Pursuit of More" as opposed to the "Disciplined Pursuit of Less." [3, 4] What is this paradox? Stated concisely, the Clarity paradox teaches that focus leads to success, success leads to opportunities, opportunities lead to diffused efforts, and diffused efforts lead to failure. The key factor in this process is *focus* – A strong focus brings clarity and success while diminished focus (or distraction) leads to diffused efforts and failure. In fact, at any point in this process, a healthy dose of focus can re-establish success (see Fig. 1) through the application of McKeown's core principles (Use more extreme criteria, Asking "What is Essential?", and Minding the endowment effect). Considering this concept, the author has come to realize that the core of these principles (focus) also applies to software engineering. Truly, many of the principles that should be followed in order

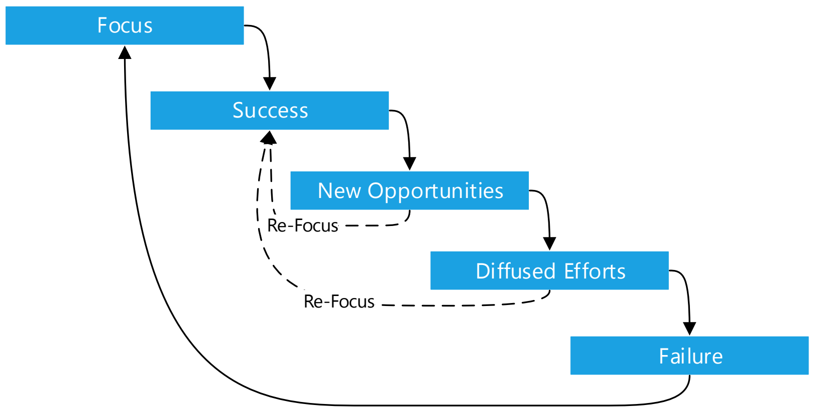

Figure 1. A graphical depiction of the Clarity Paradox as defined by

to design and develop excellent software are only a matter of good focus. Stated another way, this issue of focus is really a matter of what “could” be done as opposed to what “should” be done when designing and implementing software.

In the arena of Software Engineering, what constitutes either a good or a bad software design is a hot topic, particularly when teaching students in an undergraduate course. The differences between good and bad design are not always obvious at first (bad design almost always seem logical!), but after some pressing, students begin to pick apart flaws that have crept into an otherwise promising design, resulting in low cohesion, high coupling, and repeated code. After an in-depth conversation regarding these issues, students are brought back to the fact that they have really “discovered” two core principles that are necessary for good software design: The SRP (Single Responsibility Principle) and DRY (Don’t Repeat Yourself) [6]. After some consideration, it seems that these two core principles share a singular foundation that can be summed up in a single word: Focus. This idea has also been brought back to the forefront through a newly developed Software Engineering practice called YOGA which presents “Meditating on Code” as a key principle [7].

It is when a design or implementation loses its focus and software engineers start asking what “could” be done instead of what “should” be done that these principles are violated, typically for the sake of progress, results, time, money, and/or a lack of skill. In fact, simply adding “focus” to the development process would help to create both beautiful [1, 5] and clean [2] code! That is why, when the author begins to teach these topics, he exhorts his students to follow the core principle of Focus-driven Design/Development when they are designing and implementing software.

FOCUS DRIVEN DEVELOPMENT

Before considering these principles, consider, as an example, a potentially common code design that arises in many professional and student projects. A developer is working on a project using some language, say C++, and a namespace is created that is simply called “Utils”. As one might expect, the class is intended to hold utility functions that, quite frankly, do not fit well anywhere else. Things start off well – in this case, the class holds utility functions for loading system configurations for evaluation and study. With a small number of functions, navigating the code is easy. But soon, the class is populated with methods for converting decimal numbers to binary, manipulating strings, performing mathematics, getting timestamps, and processing command line arguments. Whether done out of convenience or ease, this results in a piece of code with diffused efforts or a lack of focus. A short list of utility functions has now become a long, mish-mashed list of functions buried in a single namespace. And now, though the code still compiles and performs well, the “Utils” namespace is filled with a conglomeration of methods and functions that must be searched and scoured every time that a method is needed.

What was the real problem in the design and implementation of this names pace (other than the hap-hazard nature of the development)? *A lack of focus!* Consider the steps that this process followed in Table 1:

1. The developer was initially *focused* and, when confronted with the need to design his software well, he created a new namespace to hold “Utility” functions. Due to his focus, he encountered *success* (In this case, a new namespace was created that held *related* functions, was easy to use, and was easy to navigate);

2. The *success* of this new design brought with it the *opportunity* to store more functions inside of the same namespace that did not quite fit anywhere else;

3. This *opportunity* led the developer to store all utility functions inside the utility namespace, leading to a code design that was no longer focused. Instead, the code contained many functions that were only loosely related as utilities and was now, in fact, *confusing* and had *lost its focus*;

4. Finally, the code is now left in a state of *failure*, where every developer has to search, scroll, and scrounge for the desired function.

Table 1. A short listing of potential thoughts that a developer may have while creating code that leads to code that lacks focus.

	Step	Developer’s Thoughts	Impact
1	Focus and clarity lead to success	“Awesome! I’ll just create a namespace to hold these extra functions even though I wasn’t sure where they should go.”	Load system models and data.
2	Success leads to more opportunity	“I really don’t want to write any extra code. I bet I can just throw this new function here and there won’t be any issues.”	A “Utils” namespace could easily hold many utility functions of all types
3	Increased opportunity leads to diffused efforts and loss of focus	“What was that function I need again?” (asked concurrently with scrolling down a long list of function definitions)	The “Utils” namespace is quickly filled with unrelated functions that perform multiple tasks that are listed in varying order.
4	Failure	“Where is that? What does it do?” (followed by scrolling and multiple uses of “Ctrl+F”)	The “Utils” namespace must be searched and scoured for each new need or change, quickly becoming difficult to work with.

This process could have easily been stopped through the addition of focus at any step in the process. For instance, if the developer simply asked, “Is this new Utils namespace *focused* on one function/process/purpose?” the answer would have quickly been found to be “no”, and, the namespace may have been refactored into “`CommandLineUtils`” or “`StringUtils`” with extra namespaces/classes being added to capture new functionality. In this case, a simple change of name would most likely have prevented this implementation from moving into a state of failure. In any case, the problem could have easily been solved by applying the principles espoused by McKeown. In terms for software development, these three principles of Essentialism can be modeled through the five principles of Focus-driven Design and Development:

1. **Focus on the location.** In the process of designing and implementing code, developers often seek the most convenient location to implement a function or method. This is the problem brought forth in the previous example. The question should never be "Where *could* I put this code?" Instead, the question should use the more extreme criteria of "What is the *best* place that I *should* put this code?" The difference is subtle, but "could" versus "should" is the fine line between convenience that leads to failure and focused effort that leads to success.

2. **Focus on the function.** The task that a method or function completes, the functionality of a class, even the logic within a simple if-statement should all be highly focused and should accomplish a single task or relate to a single model (This is commonly referred to as the SRP). If the function of the code is both correctly and highly focused, the resulting design, or form, of the code developed will be successful. It is never a question of "*What could*", but "*What should* this function do?" Perhaps the epitome of this principal is the old adage "Form follows function" or simply asking, "What is essential?"

   One objection that may rise against this idea could be, "But isn't this just recounting the SRP?" No. Simply stated, *SRP inherits Focus*. The main, driving principle behind SRP is achieving enough Focus to have the clarity of thought to use the SRP well. This principle is not recounting SRP, it is revealing the foundation and motivation behind the principle.

3. **Focus on the names.** Just as form follows function, function often follows name [1]. In other words, once a class, function, or method is named, all developers that read that code will make some type of inference and the perception of the code that "*should*" be contained within it will be forever changed. A name that is too generic – something like "`Utils`" – makes developers wonder what "*could*" fit inside, leading to a class or namespace that is filled with errata, congested with logic, and crowded with code. On the other hand, something that is more focused – like "`CommandLineUtils`" – forces developers to consider that only utility functions related to the command line "*should*" be included.

4. **Focus on Communication.** Regardless of what code does, the number one, absolutely essential responsibility that a developer has is creating code that can be maintained. This means that code must be readable and understandable to the generations of developers that will follow in their

footsteps [1]. The major question here is "What *should* this code mean to another developer?" as opposed to what "What *could* this code mean to another developer?" or "What does this code mean to me?" Peer review is a very helpful tool when asking these questions as it brings a clear, outside perspective. In short, code must clearly communicate intent and logic to a broad audience and future generations.

5. **Focus on Refactoring.** Refactoring is the key to restoring focus as it is the best time to ask questions like "What should this function do?", "Does this function or class do too much?", and "What methods should this class/namespace contain?" As developers, there should never be any fear in removing or rewriting code. One of the key, though soft, metrics of whether or not your code is focused is the Fear Factor: How afraid is another developer to refactor your code? If the major response is terror because so much of the design and the code will have to change and no one understands what the code actually does, then it is definitely time to refactor; it is definitely time to regain your code's focus. If, on the other hand, refactoring sounds like a piece of cake or a walk in the park, your code is most likely much more focused, though a good refactoring never hurt anyone.

CONCLUSIONS

In the author's opinion, a lack of focus is a major issue in modern society. When the majority of computer scientists are constantly distracted by smart phones, emails, social networks, and the like, it is difficult to truly spend time focusing on a problem or a task, thus completing it as successfully as possible. This tends to play out in while developing code – software engineers lose focus on what they should be doing and, instead, rush to complete deliverables. Perhaps software developers and engineers can take a cue from McKeown and spend more time focusing on the code.

REFERENCES

[1] Green, R., Ledgard, H., Coding Guidelines: Finding the Art in the Science, *Communications of the ACM*, 54 (12), 57-63, 2011.

[2] Martin, R., *Clean Code: A Handbook of Agile Software Craftsmanship*, Upper Saddle River, NJ: Prentice Hall PTR, 2008.

[3] McKeown, G, The Disciplined Pursuit of Less, 2012, http://blogs.hbr.org/2012/08/the-disciplined-pursuit-of-less/

[4] McKeown, G., *Essentialism: The Disciplined Pursuit of Less*, London, England: Virgin Books, 2014.

[5] Oram, A., Wilson, G., *Beautiful Code: Leading Programmers Explain how They Think*, Sebastopol, CA, O'Reilly Media Inc., 2007

[6] Pilone, D., Miles, R., *Head First Software Development*, Sebastopol, CA: O'Reilly Media Inc., 2007.

[7] David Weiss, *YOGA: A Software Development Process Based On Ancient Principles.* SIGSOFT Softw. Eng. Notes, *40 (4):5-5*, 2015.