Black Box Thinking: A New Perspective for Successful Software Development

Asem Abdelhady
Innopolis University
Innopolis, Russia
a.abdelhady@innopolis.university

Menna Awadallah
Innopolis University
Innopolis, Russia
m.awadallah@innopolis.university

Mosab Mohamed
Innopolis University
Innopolis, Russia
o.mohamed@innopolis.university

Artem Kurglov
Innopolis University
Innopolis, Russia
a.kurglov@innopolis.ru

Giancarlo Succi
University of Bologna
Bologna, Italy
g.succi@unibo.it

***Abstract*—Software development is a complex process that is constantly being optimized for over half a century. The most used approaches are focused on technical terms such as: Codebase improvement, Pipeline structure, and Integral tests coverage. Non-technical approaches have less attention in the research area, for example, pair programming and flat hierarchy. In this paper we concentrate on how the approaches driven from "Black Box Thinking"[7] book can be used in software development. By experimenting with different teams working on software projects, we measure the members' satisfaction, tests coverage, organization time, finished tasks, blocked tasks, work in progress tasks and unfinished tasks. On contrary to other approaches, "Black Box Thinking" techniques improved members' satisfaction level, decreased organizational period over time and increased test coverage.**

***Index Terms*—Software development process, Human behaviour, Software engineering, Tests code coverage**

I. Introduction

The aviation industry is a complicated field with many logistical and administrative decisions, each of which holds tremendous weight. Nevertheless, although the field has flourished over the years, it still has its fair share of failures and tough decisions, like any other industry. However, the distinctive feature of aviation is its commitment to optimization and its incremental growth mindset. As a result, aviation gained momentum and steady growth with practically non-existent catastrophes with one passenger fatality per 7.1 million air travelers [5].

In "Black Box Thinking," Matthew Syed explores the various aspects that helped aviation throughout its journey and how the slightest of decisions can help immensely guarantee the safety and happiness of the customers. The book also dives deep into concepts that can be used in various fields and industries because it strictly focuses on the human behavior aspect of life and how to think outside the box in the tamest of situations.

The term "Black Box" refers to an aircraft's black box. This device keeps various information about the flight in case the need arises to analyze that information, whether for a basic check-up or to determine the cause of a crash. In the book, Syed links the term to a way of thinking and handling situations where the sole objective of the analysis is to avoid repeating the same mistake. Syed emphasizes the importance of such an approach when dealing with anything in life, because it takes an objective view of the situation and guarantees the best outcome.

II. Problems

A. Cognitive Dissonance

Cognitive Dissonance is a term that describes the psychological state in which an individual manifests unconscious blockage of truth and denial when faced with evidence that contradicts an idea they believe in. This state can often result from an individual tying one's self-worth or status with an idea they advocate for.

It can manifest in various ways, such as lacking the ability to look at evidence and experiments in an unbiased manner. This form of cognitive dissonance is called confirmation bias: an individual may look at a diverse set of contradicting evidence, yet only draw conclusions supporting their preexisting beliefs[6].

Another form of cognitive dissonance can appear when an individual ties their self-worth with the ideas they believe in. In such a case the individual will oppose anything that might suggest the falsehood of their claims, as it might lead people doubting their character or status.

Cognitive dissonance appears everywhere in life, whether it is in students, teachers, workers, or managers. For example, cases of cognitive dissonance can be seen specifically in the medical field, where experienced doctors link every decision they make with their self-worth resulting in an environment built on arrogance and suppression instead of what people might regard as an ethical and elevated occupation.

This state can also be observed in various aspects of the software engineering field[8]. For example, the careless submission to a hierarchical company structure may lead to team leads and managers making wrong decisions for the sole fact that they are higher in the hierarchy and do not want to admit that their ideas were lacking in any way.

This behavior can be handled in a number of ways ranging from occasional workshops for leads and managers, to abolishing the hierarchy. One of the more common solutions is RCTs, that will be discussed extensively in Section III.

B. Closed Loops

Closed loop thinking describes the state where there is an individual stuck within a loop in which the "effects" feedback influences more of the "causes". The causes are often related to each other and influenced by the narrow perception of pre-established ideas of causes and effects. It obscures the bigger picture and makes it harder to get full feedback. According to

Matthew Syed, ”Closed loops are often perpetuated by people covering up mistakes. They are also kept in place when people spin their mistakes, rather than confronting them head-on. But there is a third way that closed loops are sustained over time: through skewed interpretation”[7, p. 149]. In other words, individuals would interpret data supportive of their existing beliefs or biases, rather than objectively assessing the situation.

Closed loops prevent organizations from learning from their mistakes and improving their processes. Problems, errors, or inefficiencies may go unnoticed or unaddressed, and there is no opportunity for the system to learn and improve.

This problem is often observed within software development's lifecycle when debugging system failures. Developers may often obsess over what they think is the source of a system failure, preventing them from utilizing their debugging skills and assessing every potential point of failure. There are many solutions proposed to handle this problem, varying from strategic breaks, peer programming, and up to the general idea of flexibility in development discussed in Section III.

C. Inevitability of Human Errors

The inevitability of human errors highlights the humane limitations every individual faces, despite their expertise and confidence in their work area. Individuals may always end up making mistakes[2] due to lack of sleep, lack of nutrition, or lack of hydration...etc. The idea of the ”Inevitability of Human Errors” is not meant to excuse or minimize the impact of errors, but rather to acknowledge that humans are fallible and that errors are an expected part of any complex system.

”It is still regarded as a watershed, the moment when we grasped the fact that 'human errors' often emerge from poorly designed systems. It changed the way the industry thinks” [7, p. 35], Matthew reflects. He discusses this realization that led to a greater focus on improving system design and safety procedures in the aviation industry, which has helped prevent many accidents in the years since. Accepting the inevitability of human errors is directly followed by employing critical analysis skills to system design in order to create a human error-proof system.

This is commonly seen in software development in the form of developers making typos or minor coding mistakes due to tiredness or working on repetitive tasks. Such preventable mistakes that can cause systems to fail can be fixed by introducing specific types of testing to the development process, and constantly working on identifying potential human errors. Due to such, a common solution is to focus on creating a blameless culture that prioritizes noticing how the system makes individuals behave and assessing the improvements to can work on, elaborated on in Section III.

D. The Unexpected Complexity of Life

”The Unexpected Complexity of Life” refers to understanding that many of the systems and processes in our world are far more complex and interconnected than we might initially assume. This complexity can often lead to unexpected and unpredictable outcomes, even when individuals or organizations are acting with the best of intentions. The complexity of life surpasses the human tendency to find simple solutions to complex matters. Not expecting the actual complexity of systems can lead to unfair blame when a system fails.

This concept highlights the need for humility and a willingness to learn and adapt in the face of uncertainty. It also emphasizes the importance of designing systems and processes that are resilient to unexpected outcomes and can adapt to changing circumstances.

This is a universal concept in the world of software development, where developers work hard to design a system that surpasses their grasping ability to offer innovative features. It has been often seen that managers are quick to blame and doubt the expertise of their developers, rather than acknowledge the nature of the wicked problem they're trying to solve with their software. Such a problem requires public and managerial awareness as it needs acceptance and focus on breaking down a system into small problems, improving on each part, and seeing how the system overall improves. That is, commonly called, marginal gains reviewed in Section III.

III. Solutions

A. Random Controlled Trials

Random controlled trials are one of the most effective methods for identifying the effects of introducing a new variable into an environment[3]. Such trials control the factors that are not under the direct influence of the study, while examining and trying to isolate the effects of new variables.

By conducting RCTs, organizations can systematically test and evaluate different changes or ideas to determine which are most effective for their purpose.

This approach is particularly useful in industries where managers' making wrong judgments or proposing unhelpful ideas is heavily frowned upon with serious consequences. It allows decision-makers to unbiasedly test their ideas, without confirmation bias, before integrating their ideas into the system. As such, preventing the problem of cognitive dissonance from nesting in their innovative process. RCTs promote a culture of innovation where it's expected to continuously initiate new ideas, launch tests, organize analysis, discover feedback, and integrate improvements into the system. All in all, it drives development and testing on a small scale before launching and facing adverse consequences.

B. Flexibility in Development

Flexibility in development comes from resilience to failures: the ability to expect failures and quickly adapt and change the system according to results and feedback[4]. It is when you run regular tests on your product and continue to modify it to suit your goals throughout the process of creating your product.

This is an already known strategy in the software process known as test-driven development. This strategy can be employed in multiple techniques: writing unit tests for your code, conducting user surveys, or getting frequent feedback from managers. The focus on creating exhaustive unit tests, albeit needing more effort, saves effort in debugging and fixing system problems.

It prevents developers from getting stuck in closed-loop thinking and losing their ability to identify problems in the wider scope. Furthermore, user surveys allow system creators to properly identify and modify the purpose of their system and accommodate user needs. It minimizes the risk of feeding the developer's loop of self-feedback. Lastly, getting managerial feedback, or even peer feedback, promotes thinking from different perspectives and ensures effective development.

TABLE I
PROBLEMS VS SOLUTIONS

Problems	Solutions
Cognitive dissonance	Random controlled trials
Closed loops	Flexibility in development
Inevitability of human error	Blameless culture
The unexpected complexity of life	Marginal gain

C. Blameless Culture

Blameless culture is when the work culture is accepting of failures and views them as learning experiences instead of blaming the people responsible. This type of cultures embraces failures and do not punish them because they are viewed as a way of analyzing and improving the process of development. However, this should not be confused with an 'anything goes' culture, where there is no accountability whatsoever.

These kinds of cultures focus on the potential breakthroughs that could be found when dealing with errors and failures. Such as, when the B-17 bomber in the 1940s faced multiple runway accidents that when analyzed led to the discovery of ill-designed features in the cockpit[7, p. 25].

In a blameless culture, individuals are encouraged to speak up about mistakes or near-misses without fear of retribution, which enables teams to learn from failures and continuously improve their processes. Such culture's effectiveness has been observed in software development organizations like Google's site reliability organization[1]. Their focus on errors has enabled them to look past the responsible individuals and learn the most from the downtime of services. An individual responsible for a system failure would be asked to write a detailed report explaining what happened and why, conduct meetings to provide a manual on how to avoid making that mistake again, and perhaps even present at seminars to a wider range of teams where the lessons learned are transferable.

D. Marginal Gains

Marginal gains describes the division of aspects in the development process into smaller and easier to control part, and optimizing these parts on their own. These improvements when accumulated will increase the overall performance by a noticeable amount.

Marginal gains is often used to improve safety and performance in complex systems, such as healthcare or aviation.

The Marginal Gains approach also emphasizes the importance of measuring and tracking progress, as well as learning from failures and mistakes. By continuously evaluating and making small iterative improvements, organizations can achieve ongoing improvement and ensure that they are providing the best possible outcomes for their stakeholders. When facing such complex systems, marginal gains has proved to be one of the most efficient techniques to tackle this complexity. "This is important because you must have the right information at the right time in order to deliver the right optimization, which can further improve and guide the cycle." says Vowles [7, p. 174], on how to improve Mercedes' performance in Formula one races via marginal gains.

Living in the age of data, many experts in various fields have turned into data analysis for optimisations, drawing attention to the marvelous marginal gains resulting from it. (see Table I).

TABLE II
MEASUREMENTS

Metric	Team Measurement	Individual measurement
Organizational meetings time	Average of individuals	Number of minutes
Finished tasks	Sum of individuals	Number of tasks
Work in progress tasks	Sum of individuals	Number of tasks
Unfinished tasks	Sum of individuals	Number of tasks
Tests code coverage	Average of individuals	Percentage
Member satisfaction	Average of individuals	Percentage

TABLE III
GOALS

Week	Goals
First Week	None
Second Week	Elaborate Organizational Decisions
Third Week	Communication and Knowledge Sharing
Fourth Week	Documentation and UI/UX
Fifth Week	Testing and Code Quality

IV. EXPERIMENT

The Marginal Gains strategy of Black Box Thinking attracted huge attention recently. Its application in the context of Software Development became an area of growing interest.

The basic premise of marginal gains is to focus on specific goals one at a time, and assess progress based on specific criteria that is calculated after each iteration of the process. By taking small steps to improve each individual aspect of a larger process, the hope is that over time, the cumulative effect of these marginal gains will result in significant improvements in overall performance.

In this study, we are conducting an experiment which takes the form of spike tests, where we asked six teams consisting of university students developing software to track certain metrics related to the their development process in the span of five weeks, where the first week has no specific goal for the team and is used to get an idea of the teams performance before applying the marginal gains method, and in each of the later four weeks we ask the team to pay special attention to a specific part of the development process and try to improve it as much as they can. The metrics can be seen in Table II.

The most important metric in this experiment is member satisfactions, because it is the most objective metric in terms of relating marginal gains practices to the individuals using it. Other metrics can prove difficult to analyze due to the differences between the teams and their projects, and the differences between tasks within a single project, and the many factors that could affect the development process in a non-controlled environment.

We provided the six teams with sheets of the aforementioned metrics and a specific goal for each week after the first. The goals can be seen in Table III; elaborate organizational decisions, communication and knowledge sharing, documentation and user interface and experience, testing and code quality.

A. Elaborate organizational decisions

The main objective of this goal is to ensure that the team stays on track. This is achievable by understanding each problem, dividing it into subtasks, and determining the priority of each task.

It's essential to have a clear plan of action for each task and to be proactive in identifying any potential issues that may arise, as well as developing strategies to address them effectively.

B. Communication and knowledge sharing

The idea behind the goal is to force the teams on communication optimization in a way with which they avoid all possible distractions to get the task done without extra procedures. It is mainly focused on creating a logical pipeline or a chain of information for members to acquire most of the information needed in an accessible manner.

C. Documentation and user interface and experience

The target of this phase is to get the teams to always have a product that is accessible and easy to use for all consumers, ranging from other developers to end-users.

Having this goal encourages a trial-and-error mindset, which is proven to be a beneficial method due to its focus on the user, and its flexibility creating a product that allows for quick adjustments with a low consumption of resources.

D. Testing and code quality

Finally, we wanted the teams to focus on the correctness and reusability of the code by stress tests and holding multiple code review sessions to avoid any closed-loop thinking, and discover any mistakes by receiving feedback.

This helps the teams be sure of what has already been implemented and decreases the amount of bugs that might lead to unfinished or blocked tasks in later weeks. It also brings to light some bugs or unoptimized code that can be avoided in future iterations resulting in a more efficient development process.

V. Results

A. Member Satisfaction (See Figure 1)

The goal of our experiment was to observe how focusing on marginal gains could result in the increase of member satisfaction. In the first week, it can be observed that this week had the highest level of variance and standard deviation in satisfaction levels. As the concept has not yet been standardized, some teams had the lowermost level of satisfaction, while others had a relatively high level to begin with.

Following that week, member satisfaction has witnessed a steady improvement across all teams. Particularly in week 3, teams were tasked to focus on documentation and user interface and experience, leading to a higher median and average of satisfaction. The variance in this week dropped rapidly, implying that teams focusing on the same goal had managed to collectively increase their performance in that metric. The focus in that week contributed to building to the last week with the highest average, median, and lowest variance among all weeks. All in all, as members focused on marginal gains each week, the member satisfaction improved steadily to a sustainable level across all teams.

B. Organization Time (See Figure 2)

During the first week, we observed that the organization time of different teams varied tremendously; it was potentially affected by the complexity of their project and the well structuredness of their team.

In the following week, teams were asked to focus on elaborate organizational decisions, as that was expected to help them reduce

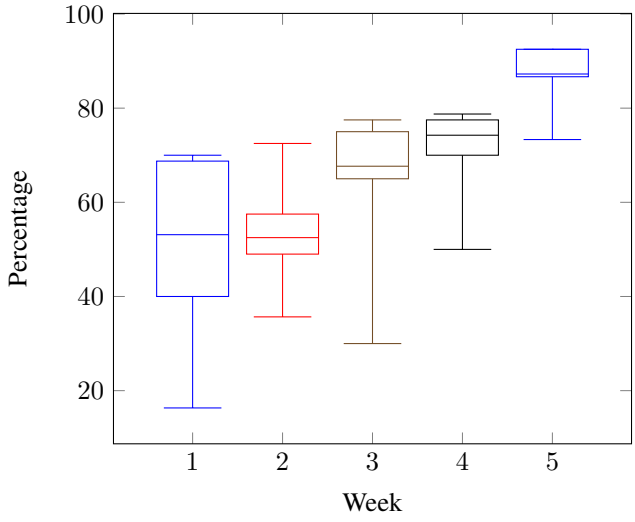

Fig. 1. Member Satisfaction

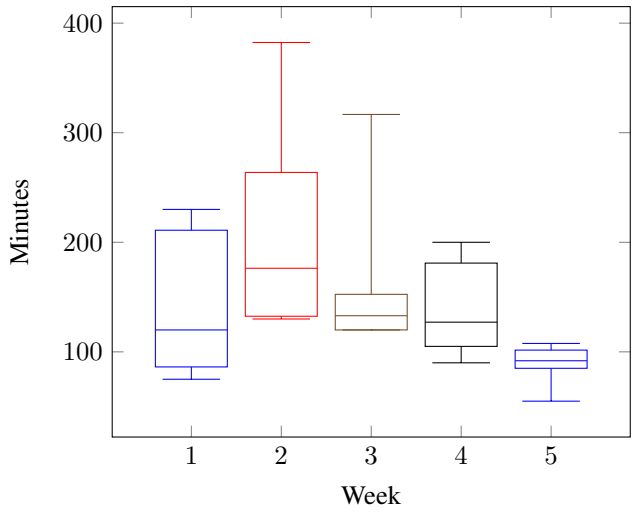

Fig. 2. Organizational Time

meetings time later on, improving that metric and increasing their productivity. As such, second week has had more variance and a higher median of time spent. It could be seen that the minimum organizational time has witnessed an increase of more than 50%.

In the following weeks, however, the majority of teams have witnessed a decrease of organizational time needed. The previous week's focus on optimizing their process and creating elaborate organizational decisions has helped decrease the time needed for the following weeks. Improving more and more each week, all teams had two weeks of relatively low organization time, converging to their local optimums depending on the complexity of their project.

Members of the experiment reported feeling more productive in meetings after redefining their structure and investing time on second week for an elaborate plan. Team leads also reported feeling like they had more time to contribute to the coding of the project now that each member was self aware and focused on tracking their metrics.

C. Tests Coverage (See Figure 3)

Testing, being one of the most important metrics in software engineering, has been taken care of since week one. As teams

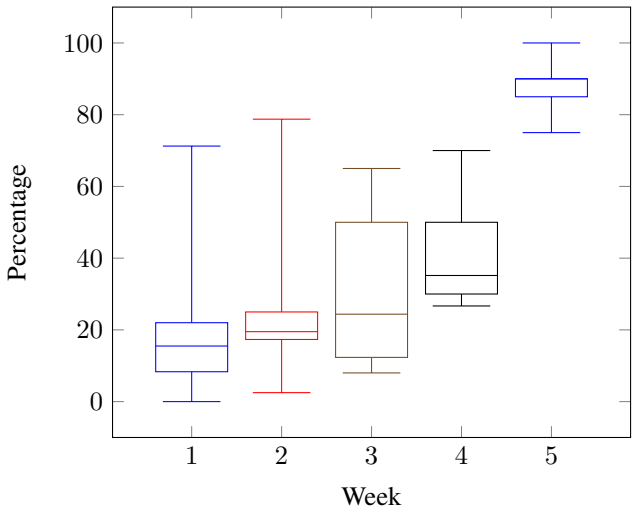

Fig. 3. Tests Coverage

focused on communication and user experience in weeks 3 and 4, their definition of tests coverage has been redefined, as well as the scope and definition of their projects. It can be observed that they have started reporting lower test coverage, despite writing more comprehensive tests. This phenomenon can commonly be explained by The dunning-Kruger effect: the more knowledge you gain, the higher is your perception of what you lack. In the last week, teams were asked to focus on improving their tests coverage. Having already optimized other metrics like organizational meetings time, teams had more time to allocate to a different priority. We have observed a significant increase in tests coverage across all teams, implying that their organization and goal definition.

D. Other metrics

Other metrics: finished tasks, work in progress tasks, blocked tasks, and unfinished tasks, were not suitable for aggregation across different teams due to the following reasons.

Firstly, the number of tasks heavily depended on teams' personal preferences and conveyed nothing about their complexity. Two members of the same team can have the same number of tasks, yet completely different complexities. So, even within one team, these metrics were not informative.

Secondly, the purpose of these metrics was not for general growth. Their purpose was personal to each member or each team in their self and team assessment process. Taking these metrics outside of their context would result in their misleading interpretability.

Lastly, optimizing these metrics has not been the goal of our experiment. Rather, the goal was to optimize the entire process of software development. All in all, other metrics proved to be hard to analyze due to their variety.

VI. Conclusion

In this paper, we identified the key problems and solutions black box thinking book discusses and analyzes thoroughly. The problems and their solutions are [Cognitive dissonance, Random controlled trials], [Closed loops of thinking, Flexibility in development], [Inevitability of human errors, Blameless culture], and [Unexpected complexity of life, Marginal gains]. We discuss how these problems manifest in the software development process while elaborating on potential helping strategies. The goal of this analysis is to improve the software development process for both the software engineer, and the customer receiving software. One of the most interesting aspects we discuss in this paper is marginal gains on conquering the unexpected complexity of life. As such, we designed a spike experiment to observe how marginal gains can improve the general member satisfaction during the software development process.

The experiment consists of five weeks, where the first week was used for calibration where there was no specific goal for that week; the reason for that was to have a control week for reference. During the other four weeks, the volunteers were given a specific area to focus on during their normal development process such as: Elaborate organizational decisions, Communication and knowledge sharing, Documentation and user interfaces and experience, and Testing and code quality.

Despite the short time of this experiment, marginal gains method showed great promise with customized software development metrics. The general member satisfaction and tests coverage has increased by the end of the experiment, while the organizational meetings time decreased, reducing the waste throughout the development process.

Albeit concluding that this experiment might not give an indisputable result, it still provides considerable insights on how the marginal gains approach can move a team closer to an optimum in a faster and more organized way. This approach is more practical than attempting to improve all aspects of the software development, which can be extremely complicated and prune to errors: due to the lack of adequate methodologies to follow. The results of this study can be the foundation of other more elaborate experiments, with a longer testing period and a greater number of volunteers from different backgrounds within the software development field researched.

References

[1] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering*. O'Reilly Media, Sebastopol, CA, April 2016.
[2] Aaron B. Brown. Oops! coping with human error in it systems: Errors happen. how to deal. *Queue*, 2(8):34–41, nov 2004.
[3] Angus Deaton and Nancy Cartwright. Understanding and misunderstanding randomized controlled trials. *Social Science Medicine*, 210:2–21, 2018. Randomized Controlled Trials and Evidence-based Policy: A Multidisciplinary Dialogue.
[4] Kent Beck et al. The agile manifesto, 2001.
[5] Clinton V. Oster, John S. Strong, and C. Kurt Zorn. Analyzing aviation safety: Problems, challenges, opportunities. *Research in Transportation Economics*, 43(1):148–164, 2013. The Economics of Transportation Safety.
[6] Uwe Peters. What is the function of confirmation bias? *Erkenntnis*, 87(3):1351–1376, April 2020.
[7] Matthew Syed. *Black box thinking*. Portfolio, November 2015.
[8] Zheng Yanyan and Xu Renzuo. The basic research of human factor analysis based on knowledge in software engineering. In *2008 International Conference on Computer Science and Software Engineering*, volume 5, pages 1302–1305, 2008.