The Importance of High Cohesion, Loose Coupling, Data Hiding and Information Hiding in Software Development

Bradley Watkins

January 2025

1 Introduction

In the realm of software engineering, the principles of cohesion, coupling, data hiding, and information hiding play a key role in determining the quality and effectiveness of software systems. Cohesion refers to the relatedness of elements within a module, emphasising the importance of keeping related functionalities together to enhance maintainability and reusability. Coupling, on the other hand, measures the dependencies between modules, with loose coupling being desirable to minimise dependencies and maximise modularity and flexibility. Data hiding is a fundamental concept in object-oriented design that involves encapsulating data to restrict direct access, thereby improving maintainability, flexibility, and security. Similarly, information hiding focuses on abstracting implementation details to reduce complexity and enhance modularity. This essay delves into these critical concepts, exploring their significance, impact on software quality, and the ways in which they contribute to the development of robust, efficient, and reliable software systems.

2 High Cohesion

Cohesion is the relatedness of elements contained within a module. Within the scope of object-oriented programming, this culminates at the class level, focusing on the relatedness of behaviour and attributes to the class's functionality [1]. High cohesion is generally considered beneficial because it leads to better maintainability and reusability of software modules. When the elements of a module are closely related, the module is easier to understand, test, and modify, which reduces the likelihood of errors and simplifies maintenance [2, 3].

A highly cohesive module generally has a single basic function and is difficult to break down into smaller,

useful modules [4]. Cohesion measures in object-oriented (OO) software design are an ever-evolving area of study with many methods being developed. The most heavily studied area of cohesion in OO systems appears to be structural cohesion, which measures information flow and the relatedness of methods within a class through their shared usage of instance variables [4, 5]. Examples of structural cohesion measures include Loose Class Cohesion (LCC) and Tight Class Cohesion (TCC), which measure the relatedness of a class's public interface, and Connectivity (C), which aims to improve on older measures by incorporating further semantic information [5].

Structural cohesion, although a well established and highly researched metric, is not able to take all aspects of software cohesion into account. For example, semantic and conceptual cohesion is lost in most structural measures and can lead to cohesive scores in an otherwise cohesive system. Measures such as Logical Relatedness of Methods (LORM) try to make use of natural language and conceptual relations within a class to provide a more nuanced cohesion score. There are also more modern cohesion measures making use of machine learning to measure software modules from multiple perspectives, providing a more accurate, holistic cohesion metric. [6].

A highly cohesive module or class is more maintainable as it is less likely to rely on other parts of the system. This means it is less likely to need updates in response to changes elsewhere in the system. Less reliance on other parts of the system also improves reusability as the integration cost of the given module will be reduced. This allows the use of the the module in other parts of the system without changing the internal workings of the module.

3 Loose Coupling

Coupling measures the dependencies between modules in a software system. Loose coupling minimises these dependencies, leading to improvements in modularity, flexibility, and maintainability, key metrics for software quality [2]. By reducing complexity within a software system, loose coupling achieves these benefits [7, 8].

Similar to cohesion metrics, coupling measurement has been a key area of research with many methods being developed. Design Structure Matrices (DSM), developed early and widely used, provide a visual representation of dependencies within a system [7, 9]. This allows engineers to map module dependencies and identify issues such as cycles and areas where dependencies can be eliminated.

The engineering approach can greatly affect the level of coupling in software of similar scope. Open source projects, where many disparate engineers work in tandem, benefit from loose coupling [8]. Increased modularity

reduces reliance between parts of the system, allowing greater agility and flexibility. There is also the benefit of changes to one part of the system being less likely to require changes elsewhere, enabling parallel work [10, 9].

Although loose coupling is generally accepted as a good design decision, aiming for no coupling within a system can lead to further problems and greater upfront cost [11, 10]. Some research suggests that moderate coupling can increase performance under certain conditions [10]. Coupling levels should be carefully considered based on goals such as performance, response time to market changes, and architectural decisions.

One approach where loose coupling is heavily considered is microservices, where an application is built as many small, scalable services [12]. These services typically perform a single task and run as their own process, allowing key system areas to scale in response to load, thus increasing performance. Microservices are highly modular, meaning systems can be swapped, updated, and removed with minimal impact on other areas due to their low coupling. However, this can sometimes reduce maintainability, as bugs can be difficult to identify and isolate with data moving between many disparate processes and increased software complexity [13].

4 Data Hiding

Data hiding is a fundamental principle in software design that involves encapsulating and abstracting data to restrict direct access from unexpected sources. In object-oriented design, data hiding is typically achieved through the encapsulation of data within a class [14]. Object-oriented languages provide access restriction mechanisms, such as the private identifier in Java, and interface definitions that serve as contracts for interacting with class data and functionality.

Data hiding offers several key benefits that enhance software quality, including improved maintainability, flexibility, and reliability [2, 14, 15]. By abstracting the data within a class, modifications to underlying data structures can be made without affecting the public interface. This separation significantly benefits maintainability and flexibility, as it minimises the impact of changes on the overall system [14, 15]. This approach also facilitates code reuse and modularity, making it easier to develop and maintain complex software systems.

Furthermore, in addition to maintainability and flexibility, data hiding enhances data security by preventing unauthorised access to sensitive information encapsulated within an object. This security aspect ensures that only authorised methods can interact with the data, reducing the risk of unexpected access and changes [16]. Studies have found that reduced complexity and improved data encapsulation in software design, both supported by the use of data hiding, contribute to improved software reliability and functionality [17].

5 Information Hiding

Information hiding, similar to data hiding, involves abstracting details from the user of a given public interface. Rather than explicitly concealing data, the emphasis is on abstracting implementation and design details through public interfaces [18]. The concept of information hiding is central to object-oriented design and offers many benefits, such as reducing complexity, improving maintainability, and enhancing modularity [19, 15, 20].

Information hiding is typically achieved through careful design of public interfaces at the class level. This can manifest as an abstract layering of conceptual ideas within the system or through the use of access modifiers to control which aspects of a given interface are available for use [20]. By carefully separating concepts within the system and exposing only necessary details, the system achieves enhanced cohesion and reduced coupling while intrinsically gaining data hiding properties. This means that a system designed with information hiding in mind will typically benefit from all the advantages associated with these concepts [20].

Maintainability is enhanced as implementation and internal design details can be changed at any time without requiring changes elsewhere in the system, as long as the public interface remains unchanged [19, 20]. The increased modularity resulting from the careful use of information hiding can also enhance the testability of a system. Definition of the public interface and understanding what a given module should achieve allows for fakes of that interface to be used in place of the true implementation. These fakes can be used as testing objects, enabling reliable unit test outcomes, or as stubs to speed up development, reducing dependency on other developers [21].

Although information hiding is a great tool for reducing complexity in a system [19], it must be carefully designed into the system. Unnecessary abstractions and low code visibility can lead to complications in automated testing [22] and reduce code readability. In spite of this, information hiding is a key concept in object-oriented programming [15] and should be a consideration of engineers when aiming for high cohesion, low coupling, enhanced data hiding and generally high quality software [2].

6 Conclusion

The principles of cohesion, coupling, data hiding, and information hiding are fundamental to object-oriented programming and a key consideration when designing high quality software. High cohesion ensured that software modules are maintainable, reusable, and easier to understand. Loose coupling minimises dependencies between modules, enhancing modularity, flexibility, and maintainability. Data hiding encapsulates and abstracts data to restrict direct access, improving maintainability, flexibility, reliability, and security. Information abstracts

implementation and design details, reducing complexity and enhanding modularity in a way that benefits maintainability and testability. These benefits result in higher quality software with reduced project costs, fewer defects, a higher likelihood of project success, and overall increased customer satisfaction [2, 3]. Engineers should carefully consider each of these principles when designing high quality software, balancing the myriad benefits with trade offs that can manifest under certain conditions.

References

[1] A. Marcus and D. Poshyvanyk, "The conceptual cohesion of classes," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 2005, pp. 133–142.

[2] F. M. Al Obisat, Z. T. Alhalhouli, T. I. Alrawashdeh, and T. E. Alshabatat, "Review of literature on software quality." *World of Computer Science & Information Technology Journal*, vol. 8, no. 5, 2018.

[3] R. Dromey, "A model for software product quality," *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 146–162, 1995.

[4] J. M. Bieman and B.-K. Kang, "Cohesion and reuse in an object-oriented system," in *Proceedings of the 1995 Symposium on Software reusability*, ser. SSR95. ACM, Aug. 1995, pp. 259–262.

[5] M. Hitz and B. Montazeri, *Measuring coupling and cohesion in object-oriented systems*, 1995.

[6] E. N. Haner Kırğıl and T. Erçelebi Ayyıldız, "Predicting software cohesion metrics with machine learning techniques," *Applied Sciences*, vol. 13, no. 6, p. 3722, Mar. 2023.

[7] D. V. Steward, "The design structure system: A method for managing the design of complex systems," *IEEE Transactions on Engineering Management*, vol. EM-28, no. 3, pp. 71–74, Aug. 1981.

[8] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the structure of complex software designs: An empirical study of open source and proprietary code," *Management Science*, vol. 52, no. 7, pp. 1015–1030, Jul. 2006.

[9] M. J. LaMantia, Y. Cai, A. MacCormack, and J. Rusnak, "Analyzing the evolution of large-scale software systems using design structure matrices and design rule theory: Two exploratory cases," in *Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*. IEEE, Feb. 2008, pp. 83–92.

[10] S. Brunswicker, E. Almirall, and A. Majchrzak, "Optimizing and satisficing: The interplay between platform architecture and producers' design strategies for platform performance," *MIS Quarterly*, vol. 43, no. 4, pp. 1249–1277, Dec. 2019.

[11] S. D. Eppinger, "Model-based approaches to managing concurrent engineering," *Journal of Engineering Design*, vol. 2, no. 4, pp. 283–290, Jan. 1991.

[12] V. Raj and R. Sadam, "Evaluation of soa-based web services and microservices architecture using complexity metrics," *SN Computer Science*, vol. 2, no. 5, Jul. 2021.

[13] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, Feb. 2021.

[14] B. Stroustrup, "What is object-oriented programming?" *IEEE Software*, vol. 5, no. 3, pp. 10–20, May 1988.

[15] M. Fowler, "Data access routines," *IEEE Software*, vol. 20, no. 6, pp. 96–98, Nov. 2003.

[16] A. Dann, B. Hermann, and E. Bodden, "Modguard : Identifying integrity & confidentiality violations in java modules," *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1656–1667, Aug. 2021.

[17] A. K. Yadav and C. L. P. Gupta, "Quantification of reliability in object-oriented software: Functionality perspective," in *2024 IEEE 1st Karachi Section Humanitarian Technology Conference (KHI-HTC)*. IEEE, Jan. 2024, pp. 1–6.

[18] S. McConnell, "Missing in action: information hiding," *IEEE Software*, vol. 13, no. 2, p. 128, Mar. 1996.

[19] P. W. O'Hearn, H. Yang, and J. C. Reynolds, "Separation and information hiding," *ACM Transactions on Programming Languages and Systems*, vol. 31, no. 3, pp. 1–50, Apr. 2009.

[20] J. Spray, R. Sinha, A. Sen, and X. Cheng, "Building maintainable software using abstraction layering," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4397–4410, Nov. 2022.

[21] H. Zhu, L. Wei, V. Terragni, Y. Liu, S.-C. Cheung, J. Wu, Q. Sheng, B. Zhang, and L. Song, "Stubcoder : Automated generation and repair of stub code for mock objects," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 1, pp. 1–31, Nov. 2023.

[22] L. Ma, C. Zhang, B. Yu, and H. Sato, "An empirical study on the effects of code visibility on program testability," *Software Quality Journal*, vol. 25, no. 3, pp. 951–978, Sep. 2016.

The Relative Merits of and Differences Between Behaviour-Driven Development, Feature-Driven Development and Test-Driven Develpment

Bradley Watkins

January 2025

1 Introduction

Throughout the history of software development, various methodologies have been proposed to improve the quality and maintainability of software systems. This essay will explore three prominent methodologies: Test-Driven Development (TDD), Behaviour-Driven Development (BDD), and Feature-Driven Development (FDD). Each of these methodologies provides a unique approach to development, each with its own benefits and challenges. Understanding the relative merits and differences of these methodologies can help teams choose the right approach for their projects and maximise their potential.

2 Test-Driven Development

Test-Driven Development (TDD) is a software development methodology where unit tests are written before the software is implemented [1]. This approach emphasizes the functional requirements of the software, ensuring that it works as intended and continues to work in response to changes, passing all tests. TDD typically follows a pattern known colloquially as "Red, Green, Refactor", in which you first write a failing test for your desired functionality (red), then write the minimal lines of code required to pass that test (green), and finally make necessary changes to the implementation while ensuring all tests continue to pass (refactor) [2]. This process places an upfront emphasis on the design of testable software and provides a sense of security when making changes to the codebase, as any changes that break existing tests are caught early [3].

Although TDD is not a new principle, it has yet to see truly widespread adoption in the software industry [4, 5]. This could be due to the upfront time investment required to write tests and the perception that TDD slows down the development process [1]. However, those who argue in favour of TDD suggest that the time saved in debugging and maintaining the codebase outweighs the time spent writing tests [3].

Improvements to software maintainability are a significant benefit of TDD, with tests acting as a form of documentation for the codebase as well as a tool for continuous integration. Some research suggests that TDD benefits overall code quality and engineering performance, not only due to the test first focus but also from improved engagement with development activities [6]. Maintaining a codebase is a significant part of the software lifecycle, with improvements to quality and maintainability offering significant benefits in cost savings and customer satisfaction [7, 8, 9].

Small to medium-sized teams and projects seem to benefit most from TDD, as the tests enhance communication between developers and provide an iterative regression suite with less need for specialised personnel [10]. However, TDD requires developers to be more versatile in their skillset, as they must write tests as well as implement the software, which may be an challenge in smaller teams [10]. Larger teams and projects, especially those that are distributed in nature, may find it more difficult to adopt TDD due to the increased complexity of communication and coordination. With features often spanning multiple components and developers, integrating and maintaining tests can become a significant challenge [11].

Overall, TDD appears to be a powerful tool for improving software quality and maintainability, with the upfront time investment in writing tests offset by the time saved in debugging and maintaining the codebase. Agile teams especially benefit from the cyclic and iterative nature of TDD, as the process fits well into the agile development lifecycle [3, 9]. Despite its many benefits, TDD is not a perfect solution, and many teams and individuals struggle to adopt the practice. The failure to adopt TDD appears to be multifaceted, with issues such as insufficient experience, lack of testing skills, and legacy code being significant barriers to success [12].

3 Behaviour-Driven Development

Behaviour-Driven Development (BDD) is a software development methodology that is an evolution of the principles set forth by TDD [13]. BDD places emphasis on higher-level functional requirements, focusing on the behaviour of the system through ubiquitous language. This facilitates collaboration between technical personnel and non-technical stakeholders, ensuring that the software meets the needs of the business [14]. Within an agile environment, BDD typically maps user stories to system behaviour following the pattern of ”given, when, then”.

The ”given, when, then” pattern in BDD is a structured way to write test scenarios, where ”given” sets up the initial context, ”when” describes the event or action taken, and ”then” specifies the expected outcome. These test scenarios can then be used to generate acceptance tests, manually or through automation, ensuring developers have clear requirements for how and when a given behaviour is complete [13].

Teams that adopt BDD typically see improvements in collaboration and communication, with stakeholders and developers working together to define the behaviour of the system [15]. The improved collaboration and involvement of stakeholders in the development process allow any misunderstandings or miscommunications to be caught early, reducing the risk of rework and ensuring that the software meets the needs of the business. There is also a major benefit in the form of living documentation, as the system's behaviour is clearly defined and documented in a way accessible to both technical and non-technical team members. Not only is this living documentation useful for onboarding and training new team members, but it also acts as a form of regression testing, ensuring that the software continues to behave as expected as it evolves [15, 14].

BDD is not without its challenges, however, as the process can be more time-consuming than traditional TDD due to the need for more training and the creation of a common language between stakeholders and developers [16, 15]. This extra training can be a significant barrier to entry for teams looking to adopt BDD, as they must convince stakeholders of the benefits of the methodology and ensure that everyone is on the same page when it comes to defining the behaviour of the system. Less experienced teams may also struggle with the abstract nature of BDD, as the focus on high-level requirements and behaviour can be difficult to define and even more difficult to translate into executable tests [16]. Larger, more mature teams inherently have an advantage in adopting BDD, as the methodology requires a significant amount of collaboration and involvement from stakeholders to be successful [15].

Despite these challenges, BDD can have a significant positive impact on the development process once it has been adopted. The key benefits of BDD include improved external quality, meaning the software fulfils the stakeholders' requirements, and enhancements to development productivity and overall software quality [13]. Improved collaboration and coordination appear to be driving forces of BDD adoption, with the methodology providing an avenue for cross-functional teams to work together to define the behaviour of the system [15, 13, 14]. The perception of BDD seems generally positive, with the methodology seeing use in a variety of development areas and with further enhancements such as timing, which allows for more complex scenarios to be defined and tested [17]. There is also ongoing work to enhance other aspects of BDD, such as increased reliability through the use of formal verification and enhanced productivity through the use of large language models [18, 19].

4 Feature-Driven Development

Feature-Driven Development (FDD) is a model-driven agile engineering methodology that emphasises delivering working features in one to three-week increments. FDD follows five basic principles: develop an overall model, build a feature list, plan by feature, design by feature, and build by feature. FDD breaks down the development process into small, manageable features, each covering a specific functionality desired by the user [20]. The implementation of FDD typically involves various roles and activities, including chief architect, chief programmer, development manager, class owner, and domain expert [21]. Due to the emphasis on roles and activities, FDD is best suited for medium to large projects, where the structure and discipline of the methodology can be fully utilised [20].

Adopting FDD can reduce complexity and enhance software quality and maintainability through continuous integration and refactoring [22]. FDD is also well-suited for projects that require a high level of scalability, as the methodology can be easily adapted to fit the needs of the project [20]. The focus on delivering working features incrementally allows for clear progress tracking and encourages collaboration and communication, ensuring the system continues to meet user needs [23]. FDD can be distilled into two main phases: discovering the feature list and implementing features one by one, with iterations focusing on building related feature packages [22]. This structured approach to development benefits teams that require clear and concise feature documentation and fits well into the agile development lifecycle [20, 22].

FDD also faces many challenges, however. The structured and disciplined nature of FDD may pose challenges for smaller teams or less experienced developers, as the methodology requires a high level of skill and experience to implement effectively [21]. In teams with less skilled developers, FDD may hinder agility and slow down the development process, as it lacks clear guidance for gathering and managing changing requirements [23, 22]. The benefits of FDD achieved by larger teams may not be realised in smaller teams, as the methodology requires a significant amount of resources and personnel that may not be available [20].

To alleviate these challenges, further research has been conducted to modify the process. Feature-driven reuse development enhances FDD by focusing on the reuse of feature sets, allowing for more efficient development and reducing the complexity of the process [21]. Simplified versions of FDD have also been proposed, which aim to reduce the complexity of the methodology and make it more accessible to smaller teams or less experienced developers [21]. Enhanced Feature-Driven Development (EFDD) includes Behaviour-Driven Development to increase stakeholder involvement and leverage the benefits of living documentation from BDD [24]. EFDD provides a clearer structure for dealing with requirements and managing changes, allowing for more flexibility and agility in the development process [24]. There is also ongoing work to present FDD as a useful methodology

outside software engineering, with applications in other fields such as systems engineering [25].

Overall, FDD is a useful tool for teams to implement an agile working strategy, with its structured and disciplined approach to development providing clear benefits in terms of software quality and maintainability. Despite the challenges faced, there are benefits to be gained from adopting FDD, especially in larger teams or projects that require a high level of scalability and clear progress tracking [20]. Further research into the modification and enhancement of FDD may provide a clearer path for teams looking to adopt the methodology, with simplified versions and enhanced features providing a more accessible and flexible approach to development [21, 24].

5 Conclusion

Test-Driven Development (TDD), Behaviour-Driven Development (BDD), and Feature-Driven Development (FDD) are three distinct software development methodologies, each offering unique benefits and challenges.

TDD focuses on writing tests before implementing the software, ensuring that the software functions as intended and remains reliable despite changes [1, 3]. BDD builds on TDD principles, emphasizing higher-level functional requirements and fostering collaboration between stakeholders and developers [13, 14]. FDD, on the other hand, is a model-driven agile methodology that emphasizes delivering working features in one to three-week increments, providing a structured and disciplined approach to development [20].

Choosing the right methodology depends on the specific needs of the project and the team. TDD is versatile and generally applicable, focusing on software quality and maintainability, which benefits teams of all sizes [6, 7, 8]. BDD is ideal for teams that require extensive collaboration and stakeholder involvement, leveraging TDD principles to produce high-level test cases and living documentation [15, 14]. FDD is particularly suited for medium to large projects that need high scalability and clear progress tracking, though it can be adapted for smaller teams [24, 21].

Importantly, these methodologies are not mutually exclusive. Teams can adopt each methodology in parallel, potentially achieving even greater benefits than using each approach in isolation [24].

References

[1] H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of the test-first approach to programming," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 226–237, Mar. 2005.

[2] P. Calais and L. Franzini, "Test-driven development benefits beyond design quality: Flow state and developer experience," in *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, May 2023, pp. 106–111.

[3] R. Jeffries and G. Melnik, "Guest editors' introduction: Tdd–the art of fearless programming," *IEEE Software*, vol. 24, no. 3, pp. 24–30, May 2007.

[4] N. C. Borle, M. Feghhi, E. Stroulia, R. Greiner, and A. Hindle, "Analyzing test driven development based on github evidence," Jun. 2016.

[5] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their ides," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE'15. ACM, Aug. 2015, pp. 179–190.

[6] W. Ren and S. Barrett, "Test-driven development, engagement in activity, and maintainability: An empirical study," *IET Software*, vol. 17, no. 4, pp. 509–525, Jul. 2023.

[7] M. Gradišnik, T. Beranič, and S. Karakatič, "Impact of historical software metric changes in predicting future maintainability trends in open-source software development," *Applied Sciences*, vol. 10, no. 13, p. 4624, Jul. 2020.

[8] S. Sakthivel, "A critical review of cost-benefit analysis for business software investments," *American Journal of Business*, vol. 38, no. 4, pp. 229–247, Oct. 2023.

[9] T. Dogša and D. Batič, "The effectiveness of test-driven development: an industrial case study," *Software Quality Journal*, vol. 19, no. 4, pp. 643–661, Feb. 2011.

[10] H. A. Ramzan, S. Ramzan, and T. Kalsum, "Test-driven development (tdd) in small software development teams: Advantages and challenges," in *2024 5th International Conference on Advancements in Computational Sciences (ICACS)*. IEEE, Feb. 2024, pp. 1–5.

[11] R. Sangwan and P. Laplante, "Test-driven development in large projects," *IT Professional*, vol. 8, no. 5, pp. 25–29, Sep. 2006.

[12] A. Causevic, D. Sundmark, and S. Punnekkat, "Factors limiting industrial adoption of test driven development: A systematic review," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, Mar. 2011, pp. 337–346.

[13] C. Solis and X. Wang, "A study of the characteristics of behaviour driven development," in *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, Aug. 2011, pp. 383–387.

[14] M. S. Farooq, U. Omer, A. Ramzan, M. A. Rasheed, and Z. Atal, "Behavior driven development: A systematic literature review," *IEEE Access*, vol. 11, pp. 88 008–88 024, 2023.

[15] L. Pereira, H. Sharp, C. de Souza, G. Oliveira, S. Marczak, and R. Bastos, "Behavior-driven development benefits and challenges: reports from an industrial study," in *Proceedings of the 19th International Conference on Agile Software Development: Companion*, ser. XP '18 Companion. ACM, May 2018, pp. 1–4.

[16] A. S. Dookhun and L. Nagowah, "Assessing the effectiveness of test-driven development and behavior-driven development in an industry setting," in *2019 International Conference on Computational Intelligence and Knowledge Economy (ICCIKE)*. IEEE, Dec. 2019, pp. 365–370.

[17] A. S. Nezhad, J. J. Lukkien, and R. H. Mak, "Behavior-driven development for real-time embedded systems," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, Sep. 2018, pp. 59–66.

[18] J. Carter and W. B. Gardner, "Bhive: Towards behaviour-driven development supported by b-method," in *2016 IEEE 17th International Conference on Information Reuse and Integration (IRI)*. IEEE, Jul. 2016, pp. 249–256.

[19] S. Karpurapu, S. Myneni, U. Nettur, L. S. Gajja, D. Burke, T. Stiehm, and J. Payne, "Comprehensive evaluation and insights into the use of large language models in the automation of behavior-driven development acceptance test formulation," *IEEE Access*, vol. 12, pp. 58 715–58 721, 2024.

[20] S. Aftab, Z. Nawaz, F. Anwer, M. Ahmad, A. Iqbal, A. Ahmad, and M. Salman, "Using fdd for small project: An empirical case study," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 3, 2019.

[21] S. Aftab, Z. Nawaz, F. Anwer, M. Salman, M. Ahmad, and M. Anwar, "Empirical evaluation of modified agile models," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 6, 2018.

[22] A. F. Chowdhury and M. N. Huda, "Comparison between adaptive software development and feature

driven development," in *Proceedings of 2011 International Conference on Computer Science and Network Technology*. IEEE, Dec. 2011, pp. 363–367.

[23] S. Ahmed, A. Razzaq, S. Zeeshan, S. Ahmed, and R. Ullah, "Evaluation for feature driven development paradigm in context of architecture design augmentation and perspective implications," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 3, 2018.

[24] Z. Nawaz, "Proposal of enhanced fdd process model," *International Journal of Education and Management Engineering*, vol. 11, no. 4, pp. 43–50, Aug. 2021.

[25] C. Granrath, C. Kugler, S. Silberg, M. Meyer, P. Orth, J. Richenhagen, and J. Andert, "Feature-driven systems engineering procedure for standardized product-line development," *Systems Engineering*, vol. 24, no. 6, pp. 456–479, Aug. 2021.