

Table of Contents

TASK 1	2
Introduction.....	2
Overview of Development Practices.....	2
Test Driven Development (TDD)	2
Behaviour-Driven Development (BDD)	3
Feature-Driven Development (FDD)	4
Comparative Analysis	5
Relative Merits	5
Test-Driven Development (TDD).....	5
Behaviour-Driven Development (BDD)	5
Feature-Driven Development	5
Complimentary benefits	5
Differences in Approach	6
Application Scenarios	7
Test-Driven Development.....	7
Behaviour-Driven Development.....	7
Feature-Driven Development	7
Conclusion	7
References	8
TASK 2	10
Introduction.....	10
Overview of practices.....	10
High Cohesion & Loose Coupling.....	10
Data Hiding	10
Information Hiding.....	11
Benefits & drawbacks.....	11
High Cohesion & Loose Coupling.....	12
Data Hiding & Information Hiding	12
Conclusion	12
References	13

TASK 1

Introduction

In modern software development, delivering quality software in the least amount of time is a big challenge. It is here that the Agile methodology comes into play, it has become one of the most adopted frameworks for managing software projects because it focuses on working in small, manageable steps, staying flexible, and encouraging collaboration. Specific practices included in Agile, like Test-Driven Development (TDD), Behaviour-Driven Development (BDD), and Feature-Driven Development (FDD), bear great responsibility in the process of improvement. They help teams stay on the same page with stakeholders, keep code quality high, and tackle different project challenges.

This report compares TDD with BDD and FDD: pros, cons, and everything else that makes these three different. The paper will decompose the key ideas, approaches to development, and where each of these approaches is applied effectively. Without getting too deep into specific tools or how to implement them, this will provide more information on the concept and where it will fit best within different projects.

It does so by first briefly explaining what each of these practices is; then, it looks comparatively at these practices in order to bring into view their peculiar strengths, differences, and how they complement each other; and finally, it concludes by discussing how to match these practices with project needs and the skills of the team, showing how they can help teams handle complex problems and build great software.

Overview of Development Practices

Test Driven Development (TDD)

Test-Driven Development is a software development methodology discovered by Kent Beck in the late 1990s. It originally gained popularity as part of the Extreme Programming methodology [1], [2], [3]. At that time, it was more commonly referred to as “Test-First” development. The basic idea behind TDD is simple: write automated test cases before writing the actual code. This ensures that no new code can be added unless there's a test for it, and it also prevents duplication in development from occurring [1].

The best way to grasp the concept of TDD is by using the “TDD mantra” that Beck has described in [2]. It is made up of three main stages:

- Red: The small test case should be written first. This would fail for it may not even compile yet.
- Green: Write the bare minimum amount of code that will make the test pass; even if it's not the “best” solution.
- Refactor (Yellow): Clean up the code. Remove duplicates and improve the design while keeping everything working as expected [2].

Aside from this iterative process, Beck also pointed out some key features of TDD:

- Short Iterations: The development cycles are extremely quick, in a matter of seconds or minutes. If a job is too big, you break it into smaller pieces.

- **Rapid Feedback:** Unit tests run almost instantly; this gives you immediate validation on any new code you write. Frameworks such as XUnit or JUnit make this easy.
- **Always Working Code:** By the end of each iteration, running tests ensure that no failures are encountered. Whatever breaks down is fixed there and then to keep the code stable.

Following these principles, TDD helps developers minimise bugs, improve general code quality, and make the development process even more effective [1], [2].

Behaviour-Driven Development (BDD)

Behaviour-Driven Development (BDD) is an Agile practice that was derived from TDD [4]. Dan North introduced BDD as an improvement on the efficiency of TDD, thus addressing some of the implications it has created, particularly when trying to collaborate between stakeholders [5], [6]. North described BDD as “a second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology” [5], [7]. While TDD focuses on writing tests before creating the actual code, BDD builds on TDD by integrating domain-driven design and object-oriented analysis. The shift in BDD is from just testing code to defining expected behaviours of the system through collaboration between technical and non-technical team members [5]. The primary objective of BDD is to create well-defined, executable software specifications while enhancing communication. In BDD, the gap between stakeholders and developers is bridged by defining and testing requirements using simple, natural language [4].

BDD describes system behaviours in a very simple, understandable way for all using the “Given-When-Then” format. This structure breaks scenarios into three parts [4]:

- **Given:** The initial state or context.
- **When:** An action or event that causes a change.
- **Then:** The expected result of that action.

These behaviour specifications are not just requirements but also forms of executable tests. In this way, it becomes easier for a team to work together and confirm system functionality during development.

The following are some key principles of BDD:

- **Collaboration:** BDD brings technical and non-technical stakeholders together to define and validate behaviours.
- **Behaviour Specification:** It focuses on describing behaviours in simple language so everyone has the same understanding of what’s expected.
- **Executable Specifications:** The scenarios written in plain language can be automated, providing quick feedback on how the system behaves.
- **Continuous Client Involvement:** Clients stay involved throughout the process, ensuring their requirements are met through clear behaviour definitions [4].

Feature-Driven Development (FDD)

Feature-Driven Development (FDD) was invented by Jeff De Luca in 1997 to address the challenges of a massive 15-month software project for a Singapore bank. The project had a 50-person team, and De Luca's framework introduced a structured, step-by-step approach to keep things manageable. It focused on modelling, listing, planning, designing, and building features, which provided a clear path for development [9].

FDD combines the principles of Agile with model-driven techniques, making it especially useful for larger teams and complex projects [10]. It's built around the idea of achieving frequent, tangible results while also supporting accurate planning, progress tracking, and measurement [10].

As explained in [10], FDD has two main phases:

- **Project Conception and Planning:** It is a phase where the actual basis of the needs of the project is determined and then goes for development. The teams have to work on the overall modelling of the project and have to develop a comprehensive list of features [10].
- **Iterative Development:** This development phase focuses on developing and delivering the features in iterative cycles, meaning at all the phases of the development process, there are deliverables that can offer value to customers [10].

Within these two phases, FDD follows five key processes [9], [10]:

- **Develop an Overall Model:** A detailed project model is created to guide all the following steps. This is done once and gives the team a big-picture view of what they're working on [10].
- **Create a Feature List:** Features are identified and organised into a hierarchy. They're prioritised based on business value and dependencies, so the most critical features are addressed first [10].
- **Plan by Feature:** A high-level plan is developed to determine the order in which features will be worked on. This includes setting milestones for the "Design by Feature" and "Build by Feature" steps [10].
- **Design by Feature:** The team creates detailed requirements and designs for each feature. These are guided by the overall model and the feature list [10].
- **Build by Feature:** Individual features are implemented and tested to ensure they meet the required standards. Once they've passed testing, they're delivered to the client [10].

By following this framework, FDD provides a structured and efficient way to manage complex projects while keeping the focus on delivering results that clients value.

Comparative Analysis

Relative Merits

Test-Driven Development (TDD)

According to [8], TDD has several advantages. It enables developers to understand the results they want to achieve and how to test them before they even start writing code. Testing and refactoring before moving to the next test can be encouraged. It encourages constant feedback through a set of unit tests and keeps the unit test data up to date and often called a “living document.” It also aids in prevention of recurring defects and makes fixing issues faster, which ultimately decreases testing time before delivery.

The benefits of TDD have been well-documented, However, [8] does point out some of the challenges with TDD that developers will face. For example, identifying when and what to test, how to confirm that the code will meet specifications, and actually knowing whether the code will provide true business value.

Behaviour-Driven Development (BDD)

Behaviour-driven development is all about aligning developers, testers, and business analysts onto a single page. In BDD, the behaviour scenarios are written in plain language, which makes it more inclusive and transparent for all stakeholders involved in the project [8]. This raises the visibility of a project whereby everyone involved will have an idea of what is taking place and how the project is faring.

One major strength of BDD is that it keeps the development focused on business values, ensuring that development efforts are heading towards organisational goals and user needs. By using language that everyone can understand, it helps avoid misunderstanding and keeps communication clear [8]. BDD also increases user satisfaction because it centres development around visible behaviours that meet user requirements. The testing predictability also gets enhanced with this focus, increasing developers' confidence. This focused style results in higher-quality code and thus should lower costs and reduce project risks over time [8].

However, BDD is not without its drawbacks. It strongly relies on the active collaboration of teams; if only testers are involved in the process, then it will surely be an unnecessary burden [8]. Although BDD itself does not require knowledge in scripting languages, modern practices in many organisations call for that knowledge anyway. Also, maintenance of BDD artefacts, such as scenarios and tests, may be quite time and resource-consuming, especially for longer projects [8].

Feature-Driven Development

Feature-driven development (FDD) is filled with several strengths that make it a great choice for software development, especially in projects needing collaboration and steady progress. According to [11], FDD therefore assists a team in delivering features incrementally, and in focusing on precisely what is directly useful to end users. Its iterative and collaborative approach, along with clear guidelines for the identification of system issues, has been ensuring that a team can adapt and improve as it goes. It also has enormous support for tracking activities. Teams can pretty easily track, by assigning progress values to features, how stuff like coding, design, and testing are going on, which is a huge help for managing parallel development. Besides, features can be delivered to the users daily or even weekly, meaning teams see very frequent tangible results.

FDD also focuses on effective communication, enhancing the collaboration between developers and clients [12]. This helps in tackling small issues early to prevent them from snowballing into bigger problems

and keeps the team on the same page. By breaking projects into manageable chunks, FDD reduces the time between finding problems and fixing them, thus cutting costs and making the process more efficient. It also goes beyond just testing code by focusing on things like coding standards, audit measurements, and other quality-focused practices, ensuring the software meets high standards from every angle.

That said, FDD does have some downsides. According to [13], its focus on predefined requirements makes it less flexible for projects where needs change rapidly. It also greatly depends on the experience of team members in crucial tasks, like feature identification, which can get pretty tricky if the team lacks such expertise. In any case, FDD really shines in environments where stability and precision are paramount, and it works wonders for teams seeking structured, scalable development.

Complimentary benefits

These strengths combined show how TDD, BDD, and FDD support each other to satisfy a wide range of development needs. TDD is very good at solidifying the understanding of requirements and maintaining quality by continued testing and refactoring of code, while BDD focuses on collaboration and aligning development to the business by involving all stakeholders and prioritising behaviours that address user needs. On its part, FDD has a strong emphasis on delivering in incremental features, effective communication, and tracking of activities in an orderly manner. It works best for big projects with stable requirements. Putting these together creates a comprehensive methodology that enforces rigorous testing, teamwork, and sure strides of progress. One will be able to achieve high-quality, user-focused software with a balance between the technical and collaborative aspects of its development.

Differences in Approach

This section describes the differences between TDD, BDD, and FDD based on principles, workflows, and results:

- TDD focuses on the technical aspect of implementing a system, while BDD focuses on the behaviours of the system [14].
- TDD is primarily developer-centric and involves the creation of tests for low-level code, while BDD is collaborative with non-technical stakeholders. It often utilises English-like languages and tools such as Cucumber and SpecFlow that define system behaviours in understandable ways for all stakeholders [14].
- TDD gives developers a clear understanding of what the system should do at the implementation level, while BDD provides both developers and customers with a better understanding of what the system should do from a user behaviour perspective [14].
- FDD approaches the matter differently by delivering customer-valued features incrementally in a structured, model-driven process [14].
- FDD also works particularly well for larger teams and complex projects, emphasising frequent tangible results and progress tracking. It differs from TDD, which is more unit-oriented in the code, and from BDD, which, though highly collaborative, is less structured in its methodology [14].

Application Scenarios

Test-Driven Development

TDD works very well in situations where code reliability is crucial, like in safety-critical systems or applications that have to be highly stable. Writing tests before the actual code helps developers in ensuring each part of the system is working as it should. The result is a reliable and easily maintainable software. This immediately shows the influence of TDD on the quality factor: for instance, projects at Microsoft dealing with TDD gave more than twice as good a level of code quality than projects developed without TDD, thus making this approach powerful to create dependable, high-quality software in industrial environments [15].

Behaviour-Driven Development

BDD is appropriate for projects that require high collaboration amongst technical and non-technical stakeholders especially. Because of the clear description of behaviours in a natural language, communication becomes straightforward; hence, it is rather appropriate for applications focused on users. A good example of this is the development of Narwhal-a classroom web app. In the provided project, the continuous interaction with the client ensured that all the features aligned with the actual needs of the users. Such iteration led to a feature-rich application that almost reached the deployment stage and proved how effectively BDD can fill the gap between developers and end-users [16].

Feature-Driven Development

FDD is best applied to large projects that have a complex requirement typically involving a long list of features. It is very appropriate for projects which require feature prioritisation and incrementality in feature development because its structured approach in feature delivery provides a lot of values to clients. For instance, it has been employed in the development of an information system for mosque management. Since FDD is capable of handling several features on the system, all requirements were met based on the specifications that the project entails, it was easy and efficient for the development process. This example shows the effectiveness of FDD in projects where detailed feature management and organised workflow are required [12].

Conclusion

Agilistic approaches like TDD, BDD, and FDD address the variability of issues created by modern software development. Each of them is based on something different: TDD pays attention to constructing dependable code by frequent testing, BDD creates open collaboration toward solutions that will provide users with their needs, while FDD performs especially well for big feature-heavy projects.

These methodologies complement each other when used together and have provided a strong basis for delivering high-quality, flexible software. By matching the methodology to the specific needs of the project and the expertise of the team, development teams can handle complexity more effectively and deliver outstanding results.

References

- [1] H. Kou, P. M. Johnson, and H. Erdogmus, "Operational definition and automated inference of test-driven development with Zorro," *Automated Software Engineering*, vol. 17, no. 1, pp. 57–85, 2010, doi: 10.1007/s10515-009-0058-8.
- [2] R. Beck, *Test-Driven Development: By Example*. Boston, MA, USA: Addison-Wesley, 2003. [Online]. Available: <https://ptgmedia.pearsoncmg.com/images/9780321146533/samplepages/0321146530.pdf>
- [3] R. Jeffries, *Extreme Programming Installed*. Upper Saddle River, NJ, USA: Addison-Wesley, 2000.
- [4] M. S. Farooq, U. Omer, A. Ramzan, M. A. Rasheed, and Z. Atal, "Behavior Driven Development: A Systematic Literature Review," in *IEEE Access*, vol. 11, pp. 88008-88024, 2023, doi: 10.1109/ACCESS.2023.3302356.
- [5] 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)*, Dubai, United Arab Emirates, 2019, pp. 365-370, doi: 10.1109/ICCIKE47802.2019.9004328.
- [6] D. North, "Introducing BDD," 2006. [online]. Available: <http://dannorth.net/introducing-bdd>. [Accessed: Dec. 30, 2024].
- [7] D. North, "How to sell BDD to the business," Skills Matter, 2009. [online]. Available: <https://skillsmatter.com/skillscasts/923-how-to-sell-bdd-to-the-business#showModal?modal-signup-complete>. [Accessed: Dec. 30, 2024].
- [8] R. K. Lenka, S. Kumar, and S. Mamgain, "Behavior Driven Development: Tools and Challenges," in *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, Greater Noida, India, 2018, pp. 1032-1037, doi: 10.1109/ICACCCN.2018.8748595.
- [9] A. F. Arbain, I. Ghani, and S. R. Jeong, "A Systematic Literature Review on Secure Software Development using Feature Driven Development (FDD) Agile Model," *Journal of Internet Computing and Services*, vol. 15, no. 1, pp. 13-28, 2014. doi: 10.7472/jksii.2014.15.1.13.
- [10] G. P. Barni, C. M. Sacchelli, and R. K. Scalice, "Agile method for product development based on C-K theory and feature-driven development," *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, vol. 46, no. 3, 2024, doi:10.1007/s40430-024-04706-3.
- [11] S. A. K. Gahyyur, A. Razzaq, S. Z. Hasan, 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 (IJACSA)*, vol. 9, no. 3, 2018, doi: 10.14569/IJACSA.2018.090334.
- [12] S. R. Riady, K. Sofi, J. Shadiq, and R. W. Arifin, "Selection of Feature Driven Development (FDD) Model in Agile Method for Developing Information System of Mosque Management," *Conference on Computer and Applied High-Performance Computing (CNAHPC)*, vol. 4, no. 2, pp. 127-136, Jul. 2022.
- [13] Z. Nawaz, S. Aftab, and F. Anwer, "Simplified FDD Process Model," *International Journal of Modern Education and Computer Science (IJMECS)*, vol. 9, no. 9, pp. 53-59, 2017, doi: 10.5815/ijmeecs.2017.09.06.
- [14] M. M. Moe, "Comparative Study of Test-Driven Development TDD, Behavior-Driven Development BDD, and Acceptance Test-Driven Development ATDD," *International Journal of Trend in Scientific Research and Development*, 2019, doi: 10.31142/ijtsrd23698.

[15] T. Bhat and N. Nagappan, "Evaluating the efficacy of test-driven development: Industrial case studies," in *International Symposium on Empirical Software Engineering*, 2006, doi: 10.1145/1159733.1159787.

[16] A. Vance and T. Cickovski, "A case study on developing a classroom web application using behavior-driven development," *American Journal of Undergraduate Research*, vol. 11, 2013, doi: 10.33697/ajur.2013.004.

TASK 2

Introduction

Among the main goals of software designers and developers has always been the development of quality software. High cohesion, loose coupling, data hiding, and information hiding are some of the design principles considered key to such quality [1]. These principles are highly influential in determining how scalable, maintainable, and reliable a software system will be. As a matter of fact, the overall quality of a system often comes down to how well these practices are implemented [2].

While these principles can also be applied in other programming paradigms, they become much more important when using object-oriented programming. OOP naturally fits the ideas of encapsulation, abstraction, and modularity. By following these guiding principles, developers can implement systems that are both easier to develop and to understand, but also will adapt easily to changes in requirements.

This report discusses high cohesion, loose coupling, data hiding, and information hiding, showing their role in software development. It gives an overview of each principle before providing a deeper look at the pros and cons of each regarding software quality. It then concludes by emphasising why these principles remain so significant in the building of reliable and maintainable systems.

Overview of practices

High Cohesion & Loose Coupling

High cohesion and loose coupling have been guiding principles for software modularisation since the early days of software engineering. First introduced in the 1970s, cohesion refers to how closely related the elements within a module are, while coupling looks at how dependent different modules are on each other [3], [4]. These two concepts are critical in building modular, maintainable software systems.

High cohesion means that the elements of a module are well integrated and related in functionality toward the performance of a specific task. It's all about internal consistency, making sure everything in the module works together toward one clear purpose [4]. For instance, a cohesive class in object-oriented programming would handle one well-defined responsibility without relying too much on other modules. This makes cohesive modules easier to understand, test, and update when necessary.

In contrast, loose coupling refers to a condition in which each module depends less on the others. Less dependence means that a change made in one part of the system would minimise chances of producing an undesirable effect somewhere else. Such flexibility can also pay off in the design phase where more simple, independent interfaces reduce the chances of later headaches [5]. In OOP, coupling often comes down to how many and how complex the connections between classes are.

Even though high cohesion and loose coupling are both highly desirable, they don't always go hand in hand. A module can be highly cohesive but still tightly coupled to other parts of the system, or it can be loosely coupled but lack internal consistency [4]. The ideal situation is when modules have both high cohesion and loose coupling, which allows developers to make changes locally without disrupting the overall system structure or maintainability [4].

These principles are not confined to object-oriented programming; they can be applied to a wide range of software architectures [4]. However, they are particularly significant in OOP, where concepts such as

encapsulation and modularity naturally meet the goals of cohesion and coupling. For example, by designing classes with well-defined responsibilities and few dependencies, developers can craft robust yet scalable systems that easily adapt to change.

Data Hiding

Data hiding is a design principle that restricts direct access to the internal data of an object; its internal state is available only through controlled interfaces. The principle of data hiding is essential for developing modular software because it reduces the likelihood that changes in one part of the system will affect another. The reason this is so, is that hiding the sensitive data structures inside some module allows for modifications of its internal details without anything externally breaking for clients [6].

Data hiding in object-oriented programming is normally facilitated through the use of access specifiers among other tools. For instance, variables are usually declared to be private so that they may not be directly accessed from outside the class. Instead, access is controlled through getter and setter methods that allow changes in the internal implementation without causing the class interaction with the rest of the system to break [6]. This is further elaborated in Appendix 1.

While the data-hiding concept has gained strong associations with OOP, the paradigm does not exclusively hold it. Actually, this principle can be made applicable in many other modular paradigms. In abstracting internal details about the inner operation inside a module, revealing to external elements only the absolutely necessary one via interfaces enhances robustness and maintainability.

Information Hiding

Information hiding was one of the fundamental concepts that David Parnas proposed for software design in 1972. The foundation for the idea of information hiding came through his seminal paper entitled “On the Criteria to Be Used in Decomposing Systems into Modules” [7]. The whole concept of information hiding is essentially based on the fact that software modules should keep their internal details private and show only that part of the information that other modules need for interaction. It also makes the modules pretty independent, simplifying the collaboration of developers dealing with different parts of the system by not stepping on each other's toes, which ultimately simplifies the collaboration and maintenance processes [8].

This principle is used in many different programming paradigms. The most obvious example, though, comes with application programming interfaces, better known as APIs. They specify a stable and neat interface for component interactions while concealing internal details. This information hiding is also supported in programming languages by means such as data encapsulation, separation of interface specification from its implementation, and the use of polymorphism to abstract out behaviours [8]. Information hiding insulates regions most likely to change, making software systems more flexible and easier to maintain over time.

Benefits & drawbacks

High Cohesion & Loose Coupling

High cohesion and loose coupling have been recognised widely as the base for maintainable and modular software systems. In other words, high cohesion provides that elements of a module are related, it is therefore more easy to comprehend, test and modify the same module itself [3], [4]. On the other hand, loose coupling minimises the interdependencies among modules, reduces ripple effects due to changes, and thus allows agile evolution of systems. The combination promotes scalability and reusability, especially in large complex systems.

Nevertheless, sometimes both are quite difficult to reach simultaneously. High cohesion modules might still have strong coupling to collaborate well with other parts of the system [4], [5]. The balance between these two principles can be achieved only by careful design, skilful developers, and following best practices, which can increase the initial development effort [5]. Too much focus on one principle over another results in lousy designs: extremely rigid systems that are poorly organised and lack focus.

Data Hiding & Information Hiding

Both data hiding and information hiding relate to encapsulation, restriction of dependencies, and improving modularity. Data hiding restricts access to the internal data of an object, most probably through private fields and accessor methods, so no internal details are unnecessarily exposed [6]. Information hiding is another technique, first introduced by David Parnas, which emphasises concealment of the implementation details likely to change. This makes the changes less affecting, and collaborative development among multiple developers easier [7]. These principles enhance the reliability and maintainability of software by protecting sensitive data and providing stable interfaces for interaction. They also reduce cognitive load on developers by abstracting unnecessary details [6], [8]. However, the effective introduction of such mechanisms for hiding usually increases design complexity. On some occasions, excessive abstraction or over-restriction of access reduces flexibility and may make it difficult to adapt the system to meet unexpected requirements. Besides, the design of APIs or interfaces following the information-hiding principle is quite effort-intensive.

Conclusion

The high cohesion, loose coupling, data hiding, and information hiding remain cornerstones of high-quality software construction. Such practices are useful to foster modularity, scalability, and maintainability by helping developers design robust and adaptable solutions. High cohesion and loose coupling make the evolution process much easier because it keeps modules consistent and reduces dependencies, while data hiding and information hiding enhance reliability by protecting sensitive data and limiting the effects of changes.

That said, effective application of these principles requires a number of considerations upfront in the design phase. Finding the right balance between flexibility, complexity, and strict adherence to best practices involves trade-offs between different aspects. Done right, these can provide a great set of building blocks for creating software that will be able to change and evolve to meet new requirements, which is why they remain very relevant today.

References

- [1] H. Izadkhah and M. Hooshyar, "Class cohesion metrics for software engineering: A critical review," *Comput. Sci. J. Moldova*, vol. 25, pp. 44–74, 2017.
- [2] D. J. Thewlis, "A Survey of Available Tools and Methods for Software Requirements Capture and Design," 1991.
- [3] M. Paixao, M. Harman, Y. Zhang and Y. Yu, "An Empirical Study of Cohesion and Coupling: Balancing Optimization and Disruption," in *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 3, pp. 394-414, June 2018, doi: 10.1109/TEVC.2017.2691281.
- [4] Z. Alzamil, "Software Coupling and Cohesion Model for Measuring the Quality of Software Components," *Computers, Materials & Continua*, vol. 77, pp. 3139–3161, 2023, doi: 10.32604/cmc.2023.042711.
- [5] V. Saxena and S. Kumar, "Impact of Coupling and Cohesion in Object-Oriented Technology," *Journal of Software Engineering and Applications*, vol. 5, 2012, doi: 10.4236/jsea.2012.59079.
- [6] M. Fowler, "Data access routines," *IEEE Software*, vol. 20, (6), pp. 96-98, 2003. Available: <https://www.proquest.com/scholarly-journals/data-access-routines/docview/215842171/se-2>. DOI: <https://doi.org/10.1109/MS.2003.1241375>.
- [7] D. L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058. <https://doi.org/10.1145/361598.361623>
- [8] S. Cleidson R B de and D. F. Redmiles, "On The Roles of APIs in the Coordination of Collaborative Software Development," *Computer Supported Cooperative Work*, vol. 18, (5-6), pp. 445-475, 2009. Available: <https://www.proquest.com/scholarly-journals/on-roles-apis-coordination-collaborative-software/docview/2258138965/se-2>. DOI: <https://doi.org/10.1007/s10606-009-9101-3>.

Appendix 1

```
public class Person { no usages
    private String name; 2 usages

    public String getName() { no usages
    |     return name;
    |
    |
    }

    public void setName(String name) { no usages
    |     this.name = name;
    |
    |
    }
}
```