

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

Justin Hendy

January 2025

Introduction

A software development methodology is a framework that shapes the process of planning, creating, testing, maintaining, and ultimately delivering software that meets the requirements defined by a user or stakeholder. A development methodology typically defines roles and responsibilities and the steps, activities, rules, or procedures required to deliver a software product. Given the variety of methodologies available within the domain of contemporary software development, each with its strengths and weaknesses, choosing a methodology for a project will be determined by how well its methods, rules and procedures meet the needs of that particular project.

Many modern software development methodologies follow the Agile Framework or adhere to the Agile Manifesto's 12 key principles. This report aims to examine three of these popular Agile development methodologies:

- Test-Driven Development (TDD)
- Behaviour-Driven Development (BDD)
- Feature-Driven Development (FDD)

Through this examination, we will explore the relative merits and differences between these methodologies and highlight the importance of understanding these differences when placed in the context of their suitability concerning differing software development scenarios.

Test-Driven Development (TDD)

Test-driven development (TDD) works on the principle of small, incremental development cycles, focusing on writing tests before writing functional code [\[1\]](#). The three key programming elements of TDD are:

- Red: Write a failing test
- Green: Write the minimum amount of code to pass the test
- Refactor: Refactor the code to improve its quality

The refactoring stage is not always necessary; however, as developers implement new features, it is essential to maintain readability and lower complexity where possible [2]. A typical, incremental and iterative cycle that implements the above red-green-refactor process is as follows:

- Write a unit test
- Run the test and see it fail
- Write the least amount of code required to pass the test
- Run the test and see it pass
- Where necessary, refactor the code to improve its quality
- Repeat the cycle for the following functionality

For new functionality to be considered complete, all previously written tests must continue to pass [2]; this is a fundamental aspect of TDD as it ensures stability, prevents new functions from breaking existing ones and provides a quick response when they do. Concerning the process provided above, Janzen et al. define a unit as being the "smallest possible testable software component" [1], and in most cases, this would be a method or function; Janzen et al. also highlight that this is occasionally up for debate, but generally within TDD, a method is the tested unit. One of the key tools for TDD is a unit testing framework, such as JUnit. JUnit is a Java-based framework and is part of the xUnit collection of frameworks. Other testing frameworks are available, such as NUnit for .NET, PHPUnit for PHP, and PyUnit for Python. The primary purpose of a testing framework is to provide a set of rules and guidelines for writing test cases, to execute each test automatically for every build, and to provide a structure for running tests and reporting results.

Behaviour-Driven Development (BDD)

Behaviour-driven development or BDD aims to develop software in a way where features or behaviours are defined first [3] via a set of specifications or requirements that are both testable and executable [4]. Each requirement consists of two components: the first part describes the feature or behaviour to implement, and the second part outlines the acceptance criteria that determine whether the desired behaviour has occurred; acceptance criteria form a scenario, and multiple scenarios can be associated with a single behaviour [5]. Scenarios are written in a human-readable format or domain specific language (DSL) and are used to define the expected behaviour of the system. Once the requirements are defined, testers will look to automate the scenarios by implementing tests based on the stated acceptance criteria, developers will write code to satisfy the scenarios, and tests will run to check that the desired behaviour/feature works correctly or as intended.

Multiple test frameworks, including Cucumber, RSpec, SpecFlow, and JBehave, have been created to support BDD implementation [4]. BDD test frameworks allow for the automation of test cases [6] by reading executable plain-text specifications of a scenario as set out in a feature file. By writing each scenario using a DSL, BDD test frameworks can understand what each scenario needs to achieve. In the case of Cucumber, the DSL employed is called Gherkin, which defines a collection of grammar rules centred around various keywords, the most important of which are Given-When-Then. Given defines the initial state of the

environment or system, when defines the action and then defines the expected result [3]. Irshad et al. provide the following example

- Given – User enters its username and password
- When – Authentication is successful
- Then – User logs into the system

In the example above, the scenario would carry a descriptive name such as "A user is able to login to the system", which is then linked to a feature/behaviour as determined by the product owner/business stakeholder [5].

Feature-Driven Development (FDD)

Feature-driven development is a model-driven approach that focuses on developing features that are small, client-valued functions within the context of a larger project. Like the other methodologies it is widely used within the field of modern software development [7]. FDD adheres to the process model ETVX or Entry-Task-Validation-Exit [7]. ETVX follows five development phases:

- Develop an Overall Model
- Build a list of Features
- Plan by Feature
- Design by Feature
- Build by Feature

The first phase, Develop an Overall Model, involves creating a high-level model that represents the system as a whole. This model reveals the system's scope, focusing more on shape than content [7]. During the second phase, developers produce a list of agreed-upon features along with the client [8]. This list is created based on the overall model and is used to guide the development process. A feature, within the context of FDD, is defined as a function with a level of business value. Feature sets are formed by grouping related features, and each feature must take no more than two weeks to complete. If a feature does not meet this criterion, sub-features are created by dividing each feature into smaller constituent parts [7]. Plan by feature produces an implementation plan for each feature focusing on assigning priority, and each feature is assigned a programmer as a class [8] with programmers referred to as "Class Owners" [7]. The fourth and fifth phases form an iterative process, where each iteration can last from one day up to two weeks [7]. Design by feature focuses on the detail of the given feature with content added to the model, eventually culminating with the production of a design package. The team members then review the design package. Once a design package is approved, the process moves to Build by Feature, and coding begins. Code inspection and unit testing also form part of this phase, and once classes are built and integrated with all previously completed features, the iteration is completed [7]. Aftab et al. list a number of key roles required for the implementation of FDD, these include a project manager, chief architect, development manager, chief programmer, class owner and domain expert.

Relative Merits and Differences

In 2006, Crispin pointed out that many agile teams had begun to embrace TDD as their software development strategy [9]; this was likely due to its iterative nature, ability to provide a robust platform for delivering working software, as previously written tests will flag when new functionality breaks existing functionality, and by placing testing before functionality it compels developers to consider different parts of the application or feature before writing any code [9]. Prior to TDD, testing would often be left until the end of the development cycle, by allowing testing to drive the design process, TDD not only informs what program code to write [10] but delivers a product that is more robust and less prone to defects, in some cases with "orders-of-magnitude fewer unit-level bugs" [9].

One of the key aims of BDD is to improve communication, shared understanding and collaboration between product owners, developers and testers. BDD seeks to achieve this state using a shared or human-readable domain-specific language (DSL) that can be used and understood by both business and technological stakeholders [11]. This approach sought to overcome traditional issues associated with agile working practices that often led to technical teams misinterpreting a stated desire of the key business stakeholders. Through their study on BDD and metric frameworks for enhanced agile practices in scrum teams, Natarajan et al. found the presence of communication gaps and collaboration deficiencies, which could hamper the ability to define useable requirements, were often cited as key issues and were particularly prevalent in the context of agile development, due to the iterative nature and need to respond to frequent product updates [11].

FDD seeks to involve the client with each iteration, from developing the overall model to inspecting a completed feature before integrating it with the main code. It is, therefore, a "client-centric development model." [7]. In addition to being client-led, FDD is iterative and focused on frequent releases of completed features [8]. FDD also has a feature-centric approach, with three stages spent refining user requirements into actionable features.

The development processes for TDD, BDD and FDD are all iterative and incremental, with many similarities. Scandaroli et al. even describe BDD as being built on the principles of TDD [12]. Crispin describes examples where the benefits of TDD allowed for greater collaboration with product owners [9], a key aim of BDD and one of the key tenets of FDD. However, there are some key differences between the methodologies. Starting with TDD, its red-green-refactor process aims to produce code that is not only rigorously tested, but the same tests drive the product's implementation. BDD replaces the starting point of TDD by creating features and scenarios written in a human-readable format to improve communication between stakeholders. With the use of BDD frameworks and a DSL, the remainder of the process is very similar to TDD, with completed code needing to meet the acceptance criteria set out in each scenario and refactoring taking place where necessary. On the other hand, FDD is a feature-centric approach, where the creation of small client-valued functions or features form detailed feature lists, which in turn drive the development process, with each feature being the purview of a software developer. Testing and refactoring will form part of the development process but are not the driving force behind it. Regarding stakeholder involvement, FDD and BDD are more aligned; however, with greater emphasis placed on planning the overall product with FDD, BDD is more focused on the individual features and their acceptance criteria. For completeness, TDD is primarily developer-led and has low non-technical stakeholder involvement.

The key advantages of TDD are its ability to produce rigorously tested code, its ability to be less prone to defects, and the fact that it allows many opportunities to refactor through

each iteration. Tests created continue to provide value as the codebase grows with each new feature. With this in mind, TDD works well in environments with frequent changes and a need for a high level of code quality. BDD provides a framework that improves stakeholder communication and collaboration and ensures high clarity regarding system requirements. Feature files and associated scenarios also provide detailed documentation of the system's behaviour. With these advantages, DD suits projects with diverse stakeholder involvement and high complexity. FDD produces predictable and reliable results with reduced risk due to the development process's small feature size and iterative nature. With a focus on client involvement, frequent releases and reliance on "lead programmers, designers and mentors" [13], large projects with a clearly defined scope will benefit from the key elements associated with FDD.

Conclusion

Although there are many similarities between TDD, BDD and FDD, given they are all under the umbrella of Agile development methodologies, focusing on iterative and incremental development, there are key differences between the three methodologies. When deciding upon a methodology for a project, key factors include the level of stakeholder involvement, the need for high test coverage, project complexity, stability of requirements and the project size. Given their similarities and Agile focus, combining the planning and client-focus of FDD with the features and acceptance criteria of BDD alongside the rigorous unit testing and flexibility of TDD could also provide a viable solution to specific software development environments.

References

- [1] D. Janzen and H. Saiedian, “Test-driven development concepts, taxonomy, and future direction,” *Computer*, vol. 38, no. 9, pp. 43–50, Sep. 2005.
- [2] R. Latorre, “Effects of developer experience on learning and applying unit test-driven development,” *IEEE Transactions on Software Engineering*, vol. 40, no. 4, pp. 381–395, Apr. 2014.
- [3] O. Bezsmertnyi, N. Golian, V. Golian, and I. Afanasieva, “Behavior driven development approach in the modern quality control process,” in *2020 IEEE International Conference on Problems of Infocommunications. Science and Technology (PIC S&T)*. IEEE, Oct. 2020, pp. 217–220.
- [4] 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.
- [5] M. Irshad, R. Britto, and K. Petersen, “Adapting behavior driven development (bdd) for large-scale software systems,” *Journal of Systems and Software*, vol. 177, p. 110944, Jul. 2021.
- [6] 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.
- [7] 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.
- [8] Z. Nawaz, “Proposal of enhanced fdd process model,” *International Journal of Education and Management Engineering*, vol. 11, no. 4, pp. 43–50, Aug. 2021.
- [9] L. Crispin, “Driving software quality: How test-driven development impacts software quality,” *IEEE Software*, vol. 23, no. 6, pp. 70–71, Nov. 2006.
- [10] D. Janzen and H. Saiedian, “Does test-driven development really improve software design quality?” *IEEE Software*, vol. 25, no. 2, pp. 77–84, Mar. 2008.
- [11] T. Natarajan and S. Pichai, “Behaviour-driven development and metrics framework for enhanced agile practices in scrum teams,” *Information and Software Technology*, vol. 170, p. 107435, Jun. 2024.
- [12] A. Scandaroli, R. Leite, A. H. Kiosia, and S. A. Coelho, “Behavior-driven development as an approach to improve software quality and communication across remote business stakeholders, developers and qa: two case studies,” in *2019 ACM/IEEE 14th International Conference on Global Software Engineering (ICGSE)*. IEEE, May 2019, pp. 105–110.
- [13] GeeksForGeeks. (2024, Nov.) Fdd full form. [Online]. Available: <https://www.geeksforgeeks.org/fdd-full-form/>

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

Justin Hendy

January 2025

Introduction

This report will explore the importance of high cohesion, loose coupling, data hiding, and information hiding in software development. Coupling describes the dependence between two or more software modules, where a module is an individual element of a computer program, such as a class, that performs a single function or completes a single task. A module's coupled state within a program is either loose (low) or tight (high). Within each module are individual elements; their level of cohesion determines the extent to which each element works toward a single aim. Elements either have high or low cohesion. Data and information hiding are techniques to encapsulate or shield data and information within modules to control program complexity and restrict direct access. These attributes all contribute to creating maintainable, secure code and lower the time and effort needed to fix issues as and when they arise. A further exploration of each attribute will now follow.

High Cohesion

This section seeks to define high cohesion and describes its key characteristics. It also discusses how software modules, and ultimately software programs, benefit from high cohesion by assessing its impact on the codebase's maintainability, readability, and testability. This section ends with a collection of scenarios that, in practice, would benefit from implementing high cohesion.

The level of cohesion between elements within a module refers to how closely they align toward a single purpose; therefore, when elements are said to have high cohesion, there is a close alignment between all elements of that module and the module has a clear and defined purpose. One measure of good software design is the presence of high cohesion throughout the codebase [1]. Modules demonstrating the key characteristics of high cohesion are "robust, reliable, reusable, and understandable" [1].

With low cohesion, code relating to a particular function or element appears in multiple modules throughout the program; this impacts the ability to understand the purpose of that function or element, as well as the modules themselves, due to the inability to find all functions of a particular module in one logical place. Similarly, to reuse a particular module or class, a key aspect of object-oriented programming, low cohesion will introduce redundant code with each class's instantiation due to the class's multipurpose nature, restricting reusability

[1]. Poornima et al. highlights that high cohesion also reduces complexity [2], where a complex system limits the ability to understand the behaviour of the broader program due to the lack of clarity within each module [3]. An increase in greater system complexity will cause an increase in the required mental effort or cognitive load needed to complete a task related to that system. With the potential for software to be inherently complex due to large requirements [2], elements must maintain high cohesion to not add to that complexity. With low cohesion, a module can become hard to debug with elements relating to that module spread across the codebase. Extending this further, unit testing and, therefore, code reliability are also impacted by low cohesion. Badri et al. have empirically studied the relationship between testing and cohesion and found a "significant" relationship between the two [4]. Ensuring that a module completes one task and that all elements within that module serve the task at hand is an example of the Single Responsibility Principle (SRP). This technique forms part of the SOLID design principle and is a tool used by developers to assist with achieving high cohesion. Examples of scenarios that would benefit from high cohesion include systems with user authentication or user creation, a library service with book borrowing or returning functionality, the reusability of a module designed to handle inputs and outputs, and an ordering service that may require functionality around creating, validating, listing, paying, and executing orders. In all scenarios, each module is a process related to another but does not need to share any functionality. Shared functionality would ultimately lead to a confusing and unreliable design.

As software systems become more complex, with changing requirements, production teams, demands, and ever-increasing lifespans, high cohesion is an important technique for enabling software developers to continue meeting these demands.

Loose Coupling

The following section will discuss the key definitions and characteristics of loose coupling, highlighting its importance in improving software design by maintaining module interdependence. It will also examine its effect on flexibility and scalability, ending with real-world examples demonstrating loose coupling in practice.

Within software engineering, coupling measures the level of dependency between modules or, in the case of object-oriented programming, classes [5]. Dependency is the level of reliance one element has on another within a software system. Alternatively, two components become coupled if at least one element influences the behaviour of another [6]. Loose coupling is a descriptor of systems where the level of connectivity and interdependence is minimal, providing increased flexibility, scalability, and improved understanding [7]. It can also minimise risk and increase maintainability. Increased scalability allows for the addition or removal of components without impacting other aspects of the system. The system is flexible as each module is isolated, allowing code refactoring, testing, and extendability [8]. Loose coupling increases resistance to errors or faults and minimises risk; when loosely coupled modules fail, they are less likely to impact others [9].

Examples of scenarios that require or benefit from loose coupling include working with databases, where separating business logic from the data access system is vital to ensure the business can adapt to its changing needs. Separating front-end and back-end components within software development teams can allow for simultaneous development, increasing the team's speed, output and confidence through separate testing packages. Ultimately, any system needing to update or change specific elements without serious repercussions for the

remaining elements will benefit from loose coupling, particularly online payment systems or plugins for music production software.

Data Hiding

This final section will discuss the key definitions and principles of data and information hiding. It will also examine the benefits of encapsulation and its implications for security and data integrity and provide illustrative examples of data hiding in software.

Data hiding is a vital practice in the world of modern programming [10], particularly concerning object-oriented programming. In essence, it hides or restricts access to data, attributes or methods contained within a class to prevent unauthorised or accidental changes to that data where the internal elements of an object are only through explicit interfaces. In many cases, the implementation of a particular object or class can be abstracted away from the end user [11]. Like the previous two topics, data or information hiding aims to lower software design complexity and increase flexibility by further separating classes by abstracting or hiding internal design and detail [12]; this encourages modularity and lowers the risk of errors whilst allowing for the reuse of code. A key principle of data hiding is encapsulation, where an object or program component containing data and the logic or procedures that use and manipulate that data encapsulate this structure through access modifiers, such as private and protected [13]. Variables, methods, and classes can have their access restricted by access modifiers. By controlling the access to these components, data hiding allows for rules to be implemented regarding the retrieval and modification of that data, allowing for improved data consistency and validation. Data hiding increases data security within a program [11], where additional methods called getters and setters allow for interaction with encapsulated data while maintaining its privacy.

Database systems will hide data in private models that only expose the data they retrieve via getters and setters. Within a banking system, it is important to hide all methods and data relating to the storage and manipulation of financial details within an object. Helpful information, such as account balances or withdrawals, can then be retrieved using external methods that have no knowledge or understanding of the internal methods contained in the object.

Information is a vital and important resource in the modern world [14], and with it, a need for greater security. Data hiding and its principles allow for the implementation of data security within the program itself. Data hiding is important to software development outside of the security benefits, and these benefits are similar to that of high cohesion and loose coupling, where modularity, reduced complexity, parallel or simultaneous development, flexibility and scalability are all byproducts of using these techniques. As modern software increases in scope, security and complexity, these tools will only continue to increase in importance.

References

- [1] M. Izadkhah, Habib; Hooshyar, “Class cohesion metrics for software engineering: A critical review,” *Computer Science Journal of Moldova*, vol. 25(1(73)), pp. 44–74, Feb. 2017.
- [2] P. U. S and S. V, “Significance of coupling and cohesion on design quality,” 2014.
- [3] M. Aiguier, P. L. Gall, and M. Mabrouki, “A formal definition of complex software,” in *2008 The Third International Conference on Software Engineering Advances*. IEEE, Oct. 2008, pp. 415–420.
- [4] L. Badri, M. Badri, and F. Toure, *Exploring Empirically the Relationship between Lack of Cohesion and Testability in Object-Oriented Systems*. Springer Berlin Heidelberg, 2010, pp. 78–92.
- [5] S. Husein and A. Oxley, “A coupling and cohesion metrics suite for object-oriented software,” in *2009 International Conference on Computer Technology and Development*. IEEE, 2009, pp. 421–425.
- [6] G. Gui and P. D. Scott, “New coupling and cohesion metrics for evaluation of software component reusability,” in *2008 The 9th International Conference for Young Computer Scientists*. IEEE, Nov. 2008, pp. 1181–1186.
- [7] C. Pautasso and E. Wilde, “Why is the web loosely coupled?: a multi-faceted metric for service design,” in *Proceedings of the 18th international conference on World wide web*, ser. WWW ’09. ACM, Apr. 2009, pp. 911–920.
- [8] F. Rademacher, S. Sachweh, and A. Zundorf, “Aspect-oriented modeling of technology heterogeneity in microservice architecture,” in *2019 IEEE International Conference on Software Architecture (ICSA)*. IEEE, Mar. 2019, pp. 21–30.
- [9] A. Lercher, J. Glock, C. Macho, and M. Pinzger, “Microservice api evolution in practice: A study on strategies and challenges,” *Journal of Systems and Software*, 2023.
- [10] Y. Cao and Q. Zhu, “Improved metrics for encapsulation based on information hiding,” in *2008 The 9th International Conference for Young Computer Scientists*. IEEE, Nov. 2008, pp. 742–747.
- [11] E. Bertino, “Data hiding and security in object-oriented databases,” in *[1992] Eighth International Conference on Data Engineering*. IEEE Comput. Soc. Press, 1992, pp. 338–347.
- [12] M. T. Valente, *Software Engineering: A Modern Approach*, Self-published, Ed. Self-published, 2024.
- [13] A. T. Cohen, “Data abstraction, data encapsulation and object-oriented programming,” *ACM SIGPLAN Notices*, vol. 19, no. 1, pp. 31–35, 1984.
- [14] R. Gupta, S. Gupta, and A. Singhal, “Importance and techniques of information hiding: A review,” *International Journal of Computer Trends and Technology*, vol. 9, no. 5, pp. 260–265, Mar. 2014.