

Abstract Testing of Software Systems

Tonislav S. Troev

Faculty of Mathematics and Informatics
Konstantin Preslavsky University of Shumen
Shumen, Bulgaria, tonislav_troeff@outlook.com

Abstract

This paper introduces the concept of abstract testing and examines its potential to address the combinatorial complexity inherent in modern software systems. Conventional testing approaches often fall short when managing complex scenarios with numerous components and dependencies. The proposed methodology offers a sustainable foundation for building more robust and maintainable test infrastructures that can scale efficiently and adapt to evolving requirements. A working implementation for .NET, the `TryAtSoftware.CleanTests` framework, demonstrates the approach in practice, generating more than 70,000 distinct test cases in a production project while achieving higher code coverage than the preceding testing practices.

Keywords: *software testing, quality assurance, combinatorics, abstraction*

1 Introduction

One day, an intriguing question crossed my mind. When learning about OOP (object-oriented programming), we are taught the importance of abstraction — we should avoid direct dependencies and instead have our classes depend on the abstraction provided by interfaces (also known as loose coupling) [1]. When writing tests, however, common practice tends to invert this principle — a test is bound to a specific class and is focused entirely on its concrete implementation.

This approach is undeniably straightforward, and for relatively simple scenarios any alternative may only introduce an unnecessary overhead. However, as scenarios grow more complex and combinatorially rich, it strongly resembles an antipattern.

This paper explores that tension and proposes *abstract testing* as a way to resolve it. The remainder is organized as follows: Section 2 examines why conventional testing becomes inefficient and costly as combinatorial complexity grows; Section 3 presents the concept of abstract testing as an elegant alternative; Section 4 discusses the practical aspects of implementing this modern idea; Section 5 concludes.

2 The Combinatorial Challenge

Consider a scenario in which tests must cover a critical piece of functionality — exporting data to persistent storage. Suppose the system supports four storage providers, each implementing the `IStorage` interface: AWS S3, Azure Blob Storage, Local Disk, and Google Drive. In addition, there are four distinct implementations of the `IExporter` interface, one per output format: PDF, JSON, CSV, and Excel (Figure 1).

The most widely adopted and practically validated approaches are the following [2]:

- **Unit testing.** Each class is tested in isolation, with all dependencies replaced by mocks. Although this approach is straightforward, the practical benefits of such tests are frequently negligible.
- **Integration testing.** The interactions between different modules of the system are tested, verifying they operate correctly together rather than in isolation — making this approach more

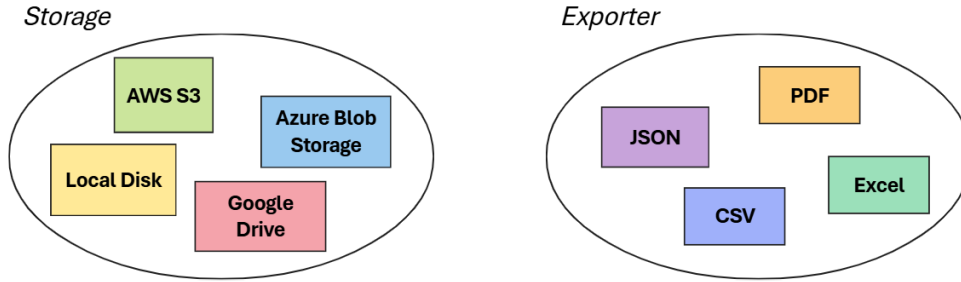


Figure 1: The sets of storage providers and exporters.

realistic than unit testing and able to catch defects the latter overlooks. These benefits come at a cost, however: establishing and maintaining a sound infrastructure is often challenging and time-consuming, particularly for complex scenarios exhibiting a high degree of combinatorial complexity.

Decisions regarding the anticipated test practices are typically made during the early stages of planning and architectural design: a project must determine which techniques and ideas are most appropriate, clarify the expectations, and commit to them [3]. Many teams follow the path of least resistance and settle for unit tests, since they are easy to write and require minimal maintenance — if the code base changes, the worst-case consequence is that some tests must be rewritten. The usual justification is that having at least some tests, despite their limited reliability, is preferable to having none at all.

This assumption, however, is worth questioning. To assess it fairly, we will look more closely at integration testing — its main strengths and weaknesses, the reasons behind its higher cost, and the means by which its challenges can be addressed.

First, each integration test requires careful data preparation and end-to-end coordination of the processes under evaluation, together with appropriate assertions at every step. Because nothing is isolated in this setting, mocks and fabricated values should be avoided; all inputs and dependencies are real. As a result, a single integration test carries more weight than several unit tests, simply because a larger portion of the code base is being exercised. For that same reason, however, a change to one component can force updates to tests whose main focus lies elsewhere. Higher-level tests that exercise several modules together are the clearest case: a test covering both the storage providers and the exporters depends on each of them, so modifying either one may break it directly.

Second, for the integration tests to be trustworthy, they must cover all — or at least the most significant — combinations of components. Enumerating every such combination amounts to taking the Cartesian product of the component sets, which in the present scenario is $Storage \times Exporters$ and already yields $4 \times 4 = 16$ distinct test cases.

The most obvious options for structuring these tests are the following:

- **Sixteen separate tests** (one per combination of storage provider and exporter). This is the weakest option: the tests are nearly identical, so they duplicate a great deal of code, and their number grows substantially with every newly introduced component.
- **Four test suites** (one per storage provider, each taking an input that supplies the exporter to use — or vice versa). This is better than the first option, but the test logic is still largely repeated from one suite to the next, and every new component requires either a new suite or an additional input, depending on which set it extends.
- **A single test suite** (taking two inputs that supply the storage provider and the exporter to use). This option removes duplication entirely and guarantees scalability — a new component requires no change to the existing logic, only an additional set of input arguments — though it is the most demanding to implement.

Assuming the final approach is implementable, the following sections will present a guiding principle for building one such setup — namely, the concept of abstract testing.

3 Abstract Testing

The concept of abstract testing emerged from the questions we posed at the beginning. It is primarily focused on addressing the combinatorial complexity inherent in modern software systems and consists of a few simple rules:

- **Test the behavior of a component, not its implementation.** A test should use random inputs rather than fixed ones, with data preparation and assertions automated wherever possible. This is what makes the approach flexible: the abstraction must describe both what a component does and the information it operates on. Heterogeneous inputs matter because they expose hidden assumptions: an implementation that works only for one specific value secretly depends on it. When this rule is followed, a test reads like an algorithm — a sequence of actions, each paired with corresponding assertions — and serves two purposes at once: confirming that the system works and documenting its capabilities and underlying processes.
- **Avoid direct dependencies within a test.** A test should refer only to interfaces; it should never inspect a component's type or name a concrete implementation. The combinatorial complexity of these scenarios makes such discipline necessary, especially when the goal is the development of a sound infrastructure. The same rule also enhances stability: new features, added components, or a reworked implementation require no change to the tests, provided the abstractions they build on stay fixed. Some auxiliary interfaces and classes are still needed for coordination purposes — for instance, to supply random data or to follow the execution plan of a given process according to specifications.

3.1 Scaling and Combinatorial Complexity

It is equally important to consider the broader picture: the test infrastructure — and, more precisely, the combinatorial setup — should remain applicable both to the current system and to its future derivatives. As noted repeatedly, the objective is a sustainable solution, one capable of accommodating further development, whether planned or as yet unforeseen. Suppose, for instance, that a new set of components responsible for compression must be introduced; it should be possible to incorporate them into the tests alongside all storage backends and exporters.

In general, the cardinality of the Cartesian product grows exponentially as new sets are added. Consequently, the number of test cases can grow uncontrollably, posing a significant challenge to the organization and maintenance of a healthy testing infrastructure.

$$|S_1 \times S_2 \times \dots \times S_n| = |S_1| \times |S_2| \times \dots \times |S_n| \quad (1)$$

$$|S_1 \times S_2 \times \dots \times S_n \times S_{n+1}| = |S_1 \times S_2 \times \dots \times S_n| \times |S_{n+1}| \quad (2)$$

Enumerating every component combination by hand does not scale; the generation of test cases must therefore be fully automated.

Although mainstream frameworks support parameterized tests, none is specialized for abstract testing. Their built-in capabilities can be utilized up to a point, but the limitations eventually become overwhelming. Several considerations illustrate this:

- **Incompatibilities.** Some components may be mutually incompatible, so that any combination containing them must be regarded as invalid. For example, if data-governance rules prohibit writing the CSV exporter's plain-text output to an external provider such as Google Drive, the combinatorial setup must be readily adjustable to reflect this constraint.
- **Dependencies.** Dependencies may exist between test components; the creation of Google Drive storage instances, for instance, may require a service to obtain an access token for the connected account of the currently authenticated user.
- **Differing lifetimes.** Certain components may need to outlive others in order to manage shared resources, such as cloud-storage clients.

These examples are representative rather than exhaustive, and further challenges of a similar nature are to be expected in practice.

4 Discussion

4.1 Tooling and Practical Results

Although abstract testing is a recent proposal, it already has working tool support. The `TryAtSoftware.CleanTests` [4] library is a test framework for .NET, built on top of `xUnit.net` [5], that generates test cases from a predefined combinatorial setup of components. It offers a range of features that support the proposed concept, including dependency injection, conditional combinability, dependency management, context sharing, and parallelization. Two companion libraries handle the data management: `TryAtSoftware.Randomizer` [6] produces the random inputs, and `TryAtSoftware.Equalizer` [7] asserts semantic equality between values. The framework has already been used in several professional projects of varying size and complexity. In one of them, it generates more than 70,000 distinct test cases that reach higher code coverage, run faster, and catch more defects than their predecessors.

4.2 Scope and Applicability

The proposed concept applies to end-to-end (E2E) testing as well. E2E tests run the system as a whole, driving it through its external interfaces exactly as a real user or client would, and thus operate at an even higher level of abstraction than integration tests. They are not treated as a separate case here because doing so would add little of conceptual value. The rules defined in Section 3 remain the same, and a combinatorial setup built for integration testing can usually be reused or extended for the purposes of E2E testing.

Abstract testing, however, demands an upfront investment — additional interfaces, orchestration classes, and a combinatorial setup must be designed before the first test case can be generated. Its value therefore scales with the combinatorial complexity: the more complex a scenario is, the more this initial cost pays off. For small, stable systems comprising few interacting components, the traditional approaches may remain the more pragmatic choice.

5 Conclusion

Abstract testing constitutes a meaningful advance in automated testing methodology. It not only provides a more effective means of ensuring the quality and reliability of software systems but also enables the development of a loosely coupled test infrastructure, thereby reducing the effort required for maintenance and modification. Further exploration and refinement of this approach are expected to clarify its potential to reshape the practice of software testing and development.

References

- [1] R. C. Martin, “Design principles and design patterns,” tech. rep., Object Mentor, 2000.
- [2] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*. Hoboken, NJ, USA: John Wiley & Sons, 3 ed., 2011.
- [3] M. Fowler and M. Foemmel, “Continuous integration.” <https://martinfowler.com/articles/continuousIntegration.html>, 2006.
- [4] TryAtSoftware, “TryAtSoftware.CleanTests: A modern framework for execution and generation of automated tests in .NET.” <https://github.com/TryAtSoftware/CleanTests>, 2022.
- [5] B. Wilson and J. Newkirk, “xUnit.net: A free, open source, community-focused unit testing tool for the .NET framework.” <https://xunit.net>, 2007.
- [6] TryAtSoftware, “TryAtSoftware.Randomizer: A library for simplifying the generation of random values in .NET.” <https://github.com/TryAtSoftware/Randomizer>, 2021.
- [7] TryAtSoftware, “TryAtSoftware.Equalizer: A library for asserting the semantic equality between two values in .NET.” <https://github.com/TryAtSoftware/Equalizer>, 2021.

Copyright © 2026 Tonislav Troev. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.