IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 9 September 2023, accepted 27 October 2023, date of publication 6 November 2023, date of current version 9 November 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3330467

RESEARCH ARTICLE

nMorph Framework: An Innovative Approach to Transpiler-Based Multi-Language Software Development

ANDRÉS BASTIDAS FUERTES[1], (Member, IEEE), MARÍA PÉREZ[1], AND JAIME MEZA[2], (Member, IEEE)

[1]Facultad de Ingeniería en Sistemas, Escuela Politécnica Nacional, Quito 170525, Ecuador
[2]Facultad de Ciencias Informáticas, Universidad Técnica de Manabí, Portoviejo 130105, Ecuador

Corresponding author: Andrés Bastidas Fuertes (andres.bastidas02@epn.edu.ec; andres.bastidas@smartwork.com.ec)

This work was supported in part by Smartwork S.A.

ABSTRACT In the rapidly evolving landscape of software development, this study aims to introduce and assess the nMorph framework, a potential breakthrough in multi-programming-language software development. This approach aspires to empower developers to code in a unified language and subsequently transpile it into multiple target languages, ensuring the software's original integrity. The objective is to balance the demands of modern software development, like adaptability, scalability, and performance efficiency, with the complexities of burgeoning digital systems. The framework endeavors to strike a balance between architectural integrity and multi-language, multi-platform flexibility. The nMorph framework, especially tailored for enterprise back-end development, integrates crucial features such as ORM, Database Connectors, a dedicated security layer, and essential tools like Configuration mechanisms and Exception Control and Logging. Architecturally, it leans on the Haxe transpiler to transition between languages like Java, C#, and PHP, and ensures compatibility with databases such as Oracle, SQL Server, MySQL, and PostgreSQL, as well as servers like IIS, Apache, Tomcat, and Glassfish. To evaluate its real-world usability, a controlled empirical experiment was conducted. Novice developers participated in a simple banking-like transactional exercise using the nMorph framework in comparison with traditional programming languages. Results from the empirical study indicated that participants, on average, took 15.19% longer when using nMorph in contrast to conventional programming languages. However, the time investment was counterbalanced by the benefits of multi-language and multi-platform software outputs using nMorph. The study also highlighted the nuanced challenges faced when transitioning to nMorph, especially contingent upon a developer's foundational language. This research illuminated the intricate interplay between perceived task difficulty, individual language proficiency, and distinct software development phases. Future directions in research will delve deeper into real-world applications, aiming for continual refinement based on user feedback and comprehensive case studies.

INDEX TERMS nMorph framework, transpiler-based architecture, enterprise back-end layers, multi-programming language software, object-relational mapping (ORM), software architecture, software development, Haxe.

I. INTRODUCTION

Software development in the modern age is characterized by a multitude of variables, including the demand for adaptability, scalability, and performance efficiency [1]. As software systems become more intertwined and complex, choosing the correct architecture is crucial. Architectural designs provide a high-level perspective, defining components and their interactions and, ensuring quality attributes such as compatibility, scalability, reliability, maintainability, and performance.

The increasing complexity of the digital world makes it imperative for foundational structures to support the continual adaptation of these systems. Structural models

The associate editor coordinating the review of this manuscript and approving it for publication was Fabrizio Messina.

perceive software architecture as a composition of components connected with aspects, such as configuration, style, constraints, and semantics, captured using Architectural Description Languages (ADLs). These ADLs simplify the depiction of components and connections and foster effective communication among stakeholders [1].

Furthermore, the practice of software design remains intricate and often result in suboptimal solutions. To address this issue, we introduce the notion of multi-programming-language software. The same software was concurrently developed in multiple programming languages. Such developments can be expensive and error-prone, necessitating the introduction of a new approach.

The concept of multi-programming language development marks a paradigm shift in the conception, design, and execution of software projects. This is further underscored by predefined architectural design models, such as model-view-controller (MVC) [2], Service-Oriented Architecture (SOA) [3], and microservices [4] which are abstract solutions for recurring software challenges.

In a previous work titled ''Transpiler-Based Architecture Design Model for Back-End Layers in Software Development'' [5], we delved into the intricacies of implementing transpilers in the back-end layer of a software architecture. That proposal aims to revolutionize the traditional models of software design by enabling the automatic transformation of business logic and back-end components from a single source code into diverse equivalents using distinct programming languages. Such innovation was not merely academic; it spoke directly to the current challenges faced by developers in addressing a dynamic, multi-platform digital ecosystem.

However, any revolutionary architectural model requires practical tools to enable its effective adoption by the developer community. Although the conceptual advantages of a transpiler-based architecture are clear, the lack of an actionable framework could potentially impede its widespread adoption. Recognizing this gap, we embarked on the journey to actualize the transpiler-based architectural vision, culminating in the ''nMorph framework.''

The nMorph framework seeks to transcend the theoretical realm and venture into a practical implementation. This toolkit has been meticulously crafted, encompassing a robust set of tools that streamlines the transpiler-based software design model [5]. Designed with the developers in mind, it provides a seamless interface to facilitate coding in a unified, translatable programming language. Such a framework enables software to be written once and, when necessary, transpiles it into various target languages, preserving the integrity and functionality of the original design.

The advent of the nMorph framework is not merely a response to the theoretical model proposed earlier but is a testament to the evolving needs of the software development landscape. Multi-programming language development is no longer a distant vision but an imminent reality, with myriad applications ranging from enterprise software suites and government projects to versatile digital platforms catering to a global audience.

This paper offers readers a comprehensive insight into the nMorph framework, starting from its foundational principles rooted in the previous transpiler-based architectural design model, to its design, features, and potential applications. Our objective is to present not only a toolkit but also a transformative methodology that has the potential to redefine multi-platform software development.

A. DEFINITIONS

As a formal definition, the ''nMorph framework'' is a cutting-edge solution in the realm of multi-programming-language software development. It allows developers to code in a unified language, which can then be transpiled into multiple target programming languages while preserving the software's original integrity. Beyond mere language translation, nMorph addresses the challenges of adaptability, scalability, and performance efficiency in modern software demands. It is particularly tailored for enterprise back-end development, integrating features such as ORM, Database Connectors, and a dedicated security layer, all underpinned by the Haxe transpiler. This framework is consolidate the balance between architectural integrity and the flexibility required for multi-language, multi-platform development.

''Multi-programming language software'' refers to software applications or systems that are developed in multiple programming languages concurrently and in their entirety. In this context, the software isn't just incorporating components from different languages, but rather, has complete parallel implementations in various languages. Each of these implementations encompasses all features and functionalities, allowing the software to be available in different programming languages, capitalizing on the unique strengths and capabilities of each language.

B. OBJECTIVE

This study introduces the nMorph framework, a groundbreaking design aimed at transcending traditional boundaries. By leveraging the power of transpilers, nMorph promises a unified business logic layer that can be transpiled across multiple programming languages, thereby ensuring functionality consistency. This approach not only simplifies the development process but also enhances maintainability and adaptability in the future.

Modern enterprises requires software platforms that are adaptable, maintainable, and future-proof. To address this need, we present a pioneering approach to software architecture-using a high-level transpiler language for crafting the business logic layer of enterprise systems. This approach ensures that the logic layer can be transpiled seamlessly across various target programming languages. Consequently, equivalent source code components emerge, each in a distinct programming language but are functionally analogous.

C. SCOPE

The crux of our proposal is not just about transpilation. Beyond this, the framework requires the following:

- Generic libraries for consistent data types and cross-cutting methods.
- Diverse database connection methods, eliminating the need for re-writing code.
- Differentiated execution mechanisms for server applications, ensuring native performance.
- Object transference through code generation to aid user interface integration.
- Post-compilation methods, artifact adaptation, post-transpilation code injection, and package generation.
- In the following sections, each component will be elaborated upon, providing an in-depth understanding of their integration and operation.

D. RATIONALE

Given the multi-platform capabilities of languages such as Java (via JVM) [6] and .net Core [7], one might question the relevance of our proposal. Additionally, with cloud deployments increasingly obfuscating back-end intricacies and protocols such as HTTP fostering interoperability, why pursue multi-language compatibility?

We posit that, while many languages offer multi-platform support, specific application scenarios demand source code in multiple languages simultaneously. While integration across platforms is feasible, there are several reasons for validating our approach:

- Customization: Enterprises often prefer specific programming languages and databases aligning with their current technological investments.
- Technological Longevity: Enhancing the software's future relevance without reprogramming presents significant long-term cost savings.
- Brand Independence: Our model offers freedom from global brands or service providers, enabling software migration between technologies without re-writing.
- Foundation for Future Research: Our multi-language programming approach lays the groundwork for other related research areas.

E. CONTRIBUTION

Traditionally, software architects have limited options and often bind them to long-term dependencies. Redundant programming across languages is impractical. The nMorph framework challenges the status quo. By using a transpiler for business logic programming, developers write their solutions once, transpile them effortlessly to various languages. Such flexibility promises broader compatibility and structuring, primarily focused on back-end development, given the ubiquity of front-end technologies, such as HTML.

Although the study identifies specific application scenarios substantiating our design, it does not restrict other solutions from harnessing the proposed benefits. Importantly, our focus is not on creating a new transpiler but on utilizing existing ones to structure this innovative software architecture design and affirm its feasibility.

F. RESEARCH QUESTIONS

- In the context of multi-programming language software development, how does the transpiler-based approach of the nMorph framework demonstrate feasibility and efficacy compared to traditional programming languages?
- How does the implementation of the nMorph framework influence the experience and efficiency of developers, especially in comparison to commonly used programming languages (C#, PHP, or Java) for business logic development?
- Considering the transition between the nMorph framework and traditional programming approaches, to what extent does a developer's language proficiency impact the adaptability, scalability, and performance outcomes in software construction and practice?

G. ABOUT THIS WORK

In the subsequent sections, we delve into the design principles of the nMorph framework, its core components, and its unique capabilities. Practical examples elucidate how the framework can be deployed in diverse scenarios, underlining its versatility and robustness. We also explore the synergies between the previously proposed transpiler-based architecture design model and the nMorph framework, highlighting how the latter actualizes the former's vision. Finally, we reflect on the broader implications of this toolkit for the future of software development, setting the stage for further research and exploration in this intriguing domain.

In this document, the term ''transpiler'' is prominently utilized, alongside several associated terms that, in essence, convey the same meaning. These synonymous terms encompass the transpiler, transcompiler, source-to-source compiler, s2s compiler, and cross-compiler. Throughout the diverse sections of this paper, the term ''translation'' is employed to denote syntax transformations. Nonetheless, it is imperative to highlight that this term is not formally recognized as a direct synonym. Its usage can be ambiguous, potentially overlapping with domains outside computer science, such as linguistics or education.

Although various methodologies exist, including automatic code generation techniques, template-based code generation, domain-specific languages, and even software generation based on artificial intelligence methodologies, which can generate outputs across multiple programming languages, this study specifically focuses on the application of a transpiler. This is viewed as the central component of the novel software architecture design paradigm. Prospective research endeavors may integrate or amalgamate these techniques to enhance the design model delineated herein.

This investigation does not venture into the crafting of a novel transpiler or delve deeply into the intricacies of the source code or the syntax translation procedure. Rather, its emphasis lies in defining design elements that leverage

the advantages of integrating a transpiler within the coding process and execution pipeline of a transactional application.

As elucidated in this document, the phrase 'multi-programming-language software' alludes to the practice of constructing software by concurrently employing multiple distinct programming languages throughout the software development lifecycle.

The remainder of this study is structured as follows. In Section I, the introductory section (currently under discussion) provides an overview of the background, objectives, scope, rationale, contributions, and research questions. In Section II, we delve into the intricacies of the nMorph framework implementation, elucidating its components, technology selection criteria, overarching development framework, and resultant solution artifacts. Section III is dedicated to an empirical experiment, a laboratory exercise undertaken by a developer team utilizing the aforementioned framework. Section IV engages in a comprehensive discussion, encompassing a review of the research questions, an analysis of previously published articles by the authors, and potential threats to validity. Finally, Section V draws conclusions from the presented content and provides avenues for future research.

II. nMorph FRAMEWORK

A. FRAMEWORK VISION

A novel application development framework is proposed, called ''nMorph framework,'' enabling the construction of enterprise back-ends using transpilable languages, which, after transpilation and integration, ensures consistent operational performance across various platform-specific compilations, like .net DLLs or Java's JAR/WAR.

This section delineates the componentes of the proposed framework, designed for high-level functional replication in various scenarios, with an architectural model based on a layered pattern to comprehend the required components and delineate their responsibilities and interactions.

The proposed development framework serves as a comprehensive library for software developers, providing the essential components required to build an enterprise platform or information system. The library is constructed in the transpiler language to ensure universal compatibility across various technologies. Among the core components are ORM (Object-relational Mapping) & Database Connectors, which bridge the application's logic with relational databases through unified connectors, abstract the specifics of database engines and promote a single high-level framework for all technologies. The framework also includes a security layer the offers methods for encryption, authentication, authorization, and data validation, ensuring that developers adopt secure programming practices.

Additional features include Common Methods, providing routine functions, such as serialization, data structures, and business logic utilities. A Configuration mechanism, typically using encrypted text-based files, allows the setting operational of parameters, such as database connection strings, file paths, logging preferences, and business logic parameters. Exception Control & Logging unify error handling and logging across platforms, ensuring comprehensive error data recording for technical review and operational transparency. Finally, Runtime Methods enable compatibility of business logic methods across platforms with foundational classes and a centralized routing mechanism for external invocations, ensuring consistent communication with back-end proxies.

The Data Layer in the proposed architecture facilitates the developer's data access logic, acting as a bridge to the underlying business logic. It encompasses a Database Engine that integrates various transactional database engines, enabling users to select their preferred database systems. This layer allows data storage and query processing by utilizing database objects, such as tables, views, and procedures.

Notably, developers can embed business logic directly into a database via stored procedures, which limits portability across different database engines in the native database language. Hence, to ensure seamless database engine transitions, the use of database-specific stored procedures should be minimized, and ORM could support DDL in the transpiler, enabling automatic generation of database logic objects. The Business Data Access component introduces programming objects mirroring database entities, with columns translated to fields in code and relationships interpreted as arrays from related tables, all leveraging ORM's standardized query methods.

The Business Layer of the architecture is designed for programmers to implement specific solution methods directly, making it the primary interface for source code integration. At its core, the Business Logic component allows developers to embed and customize algorithms that are essential for the functioning of enterprise platforms or information systems, setting business rules and conditions using business objects, and the data access layer. Business Objects provide a means to create compound or extended data structures necessary for processing information in business logic or for input and output parameters.

The Business Services component auto-generates the exposure of specific methods flagged in the business logic, acting as the interface for nested implementations, integrating calls to the framework to manage routing and handle requests throughout their lifecycle, and serving as the sole connection point for these implementations.

Nest Implementations refer to the various methods employed by target technologies to expose back-end functionalities to the front-end. These transpiled functionalities can be expressed through multiple nest implementations using single source code. For instance, with C# as the target transpilation language, business logic methods can be exposed using different technologies such as XML Web Services, WCF, Remoting, WebApi Controllers, and REST Services. Despite distinct foundational technologies, these methods maintain equivalent business logic. The greater the variety of nest implementations incorporated into the

framework, the more adaptable the software is to different operating platforms.

Additionally, as the transpiler expands to support new programming languages and specific nest implementations for them, existing source codes can be recompiled into the technology with minimal programming effort. While one might assume that nest implementations solely revolve around HTTP communication compilation or specific architectural patterns, they can also encompass areas such as asynchronous batch processes based on schedules, TCP communication, or cloud functions for serverless schemes, such as Azure Functions or AWS Lambda, which can handle both HTTP requests and event-driven tasks.

The Presentation Layer, while primarily focusing on back-end development, integrates the key front-end elements. Within this layer, the invocation framework is incorporated to facilitate serialization, message packaging, exception handling, and service invocation, ensuring compatibility with the exposed back-end services. Moreover, methods equivalent to the exposed business logic are generated for easy front-end invocation, and objects used for input and output parameters are developed in the programming language of the user interface, with multiple proxy versions tailored for various technologies. In addition, front-end developers handle the user interface layer, embedding specific presentation logic and data manipulation while utilizing proxy methods to display data or initiate transactions, thereby ensuring a cohesive system operation.

Cross-component tools play a crucial role in the system architecture. PostBuild Utility is pivotal for compiling artifacts specific to each execution technology, spanning the entire back-end layer and operating based on the Framework Concept process, primarily focusing on the sequential execution of the Preparation Process, integration of the runtime framework, and generation of back-end proxies. When employing the database-first strategy, where database design precedes programming, the DB Gen component becomes essential. It can automatically read the database table and view structures, subsequently generating the necessary programming in formats compatible with the generated ORM. Each database engine supported in the generation requires a distinct version of the DB Gen.

Note that not all the components described in this section will be implemented in the initial version. Their incorporation may have proceeded gradually.

B. TECHNOLOGY SELECTION

The nMorph framework is a reference implementation that puts the architectural design model proposed in previous studies into practice. This implementation is beneficial for determining the feasibility of the proposal and for measuring the operational capacity that can be achieved, aiding evaluations and experimentation in real-world scenarios. The criteria used for reference implementation were as follows:

1) THE TRANSPILER

The core component of the proposed architectural approach is the transpiler choice. Various transpilers in the market serve specific purposes [8], with common uses sich as converting linear algorithms to parallel ones; transpiling front-end JavaScript variants such as Typescript, CoffeeScript, and Dart; aiding in the migration of legacy systems to new specific languages; adjusting industrial electronic circuits, and other unique scenarios. A few years ago, a transpiler named Haxe was introduced [9], and in this study, it boasted the broadest compatibility with popular programming languages in the enterprise application development market. Haxe supports transpilation to C#, Java, C++, Python, and PHP, among others, thereby making it suitable for the envisioned objectives.

Haxe is an open-source transpiler accompanied by a suite of tools and a runtime environment. It translates source code written in the Haxe programming language, which is object-oriented and strictly typed, to any of its target languages in a ''one-to-many'' fashion. Originally designed for multi-platform game development, Haxe's application expanded over time to support various programming languages, even though its primary usage today is not for enterprise applications. It allows for algorithmic translation that retains the object-oriented structure in the target language, while integrating unique execution features and ensuring syntactic equivalence between languages.

Given its compatibility with the languages frequently employed in enterprise applications and its versatility in source code translation, Haxe was chosen as the transpiler for the development of the proposed framework. Future iterations of this research might utilize different transpilers to produce reference implementations using alternative technologies.

HaxeLib, Haxe's library management tool, was considered for its essential libraries, which are pivotal to the project's goals. Key libraries include hxjava for Java transpilation, hxcs for C#, hxcpp for C++, haxe-crypto for encryption and security support, and hscript for runtime scripting, among other complementary libraries.

2) DEVELOPMENT IDE

Similar to many other programming languages, Haxe structures its classes and methods within text files. Consequently, several tools assist developers in coding using Haxe by offering contextual aids, visibility of classes and methods, and code generation. For this project, HaxeDevelop was chosen because it focuses on this specific language. HaxeDevelop, a fork of the FlashDevelop IDE, offers a unified project structure and file management system, making it ideal for comprehensive compilation of components in a singular sequence. Its visual assistance provides developers with an edge, particularly when implementing a business logic layer. Although the solution was designed using HaxeDevelop, developers can utilize any IDE of their choice, because the compiler operates as a command-line program, callable from any tool, or even manually.

3) TARGET LANGUAGES

For the reference implementation aimed at determining the operational feasibility of the referred architectural design model, the selection of the target languages that best enable testing and experimentation is crucial. At the time of the study, Haxe supported transpilation to languages such as Javascript, C++, C#, Java, JVM, Python, Lua, PHP, Flash, Hashlink, and NekoVM. Three languages were chosen from this list.

- Java was selected because of its widespread adoption across devices and servers, facilitated by its virtual machine scheme which is compatible with almost all platforms, including cloud environments. They are extensively used for back-end development in enterprise applications.
- C# was chosen for its comprehensive support of Microsoft technologies, which are widely adopted, especially in the business sector and cloud. Also .Net Core is considered because it offers compatibility across various platforms beyond the Microsoft ecosystem, and is a common choice for enterprise back-end application development.
- PHP, an open-source native tool that runs in the interpreted mode, is widely used for dynamic web projects without inherent licensing costs. While it is not typically used for back-end application development because of its web-centric nature, it was chosen for this project to demonstrate its potential equivalence in back-end execution to other selected languages.

Although other supported languages, such as Javascript with Node.JS or Python with Flask, are also prevalent and versatile, this study limits its scope to the aforementioned languages. Further research should explore other languages.

4) DATABASE ENGINES

For reference implementation on the nMorph platform, it is essential to ensure compatibility with widely used enterprise relational database management systems (RDBMS). The chosen databases included the following:

- The Oracle Database [10], renowned for its robustness, stability, and pioneering data management capabilities, is a top choice among businesses owing to its vast compatibility and comprehensive support.
- The Microsoft SQL Server [11], another widely adopted RDBMS in the business sector, is known for its transactional support, scalability, user-friendly graphical environment, and performance. Recent versions have supported non-Windows platforms.
- PostgreSQL [12], a free open-source database, has recently gained popularity owing to its use in open-source software projects and performance flexibility.
- MariaDB [13], another open-source solution, is lightweight, user-friendly, and provides vast community support. By supporting MariaDB, there is inherent support for MySQL [14], its licensed counterpart.

While other relational and non-relational databases are gaining traction in the enterprise landscape, this study limits its focus to these databases. Future studies should consider supporting other databases.

5) APPLICATION SERVERS

The first version of the nMorph platform supports various application servers contingent on target transpilation technology:

- For the .net framework, given its Windows OS dependency, the applicable application server is the Internet Information Services (IIS), supporting only the versions above .net Framework 4.0.
- For .net Core, both IIS and the multi-platform Kestrel can be employed, running on Windows, Linux, or MacOS, considering versions 3.1 and 5.0.
- Java technologies are compatible with several application servers and JDKs, including:
  - Tomcat 10 with JDK 11, Tomcat 9 with JDK 8.
  - TomEE 9 with JDK 11.
  - Glassfish versions 5.1 and 6.1 with JDKs 8 and 11, respectively.
  - WebLogic, WebSphere, and Wildfly 24, all with JDK 8. Java application servers can run on various operating systems, including Windows, Linux, and MacOS.
- For PHP, the primary support is for the Apache + PHP combination. However, it is adaptable as a component on numerous application servers, such as the IIS with FastCGI. Only PHP versions above 7.0.0 are supported, aligning with the servers' supported versions. Depending on installation compatibility of the web server, PHP can run on Windows, Linux, and MacOS.

Although these are verified application servers from the research project, the compiled product might also function on other platforms with similar execution parameters.

C. FRAMEWORK BUILDING PHASE

The implementation process involved constructing a software solution based on the Haxe compilation architecture, while adhering to all aspects and components of the transpiler-based design model [5]. This necessitated the creation of several software subprojects to implement each component.

To initiate this process, the layered design diagram defined in the transpiler-based design model was referenced, which determined each stage of the tool construction. The initial implementation serves as the foundation. Thus, a framework construction project was planned and executed.

Figure 1 illustrates layers and components implementation diagram. Libraries including nMorph.common, nMorph.esb, nMorph.orm, and nMorph.svc were developed for the development environment layer, following the conceptual guidelines set out in the previous section. This endeavor was time-consuming because of the extensive technical specifications laid out for these components, particularly

in constructing the ORM and profiling the corresponding connectors.

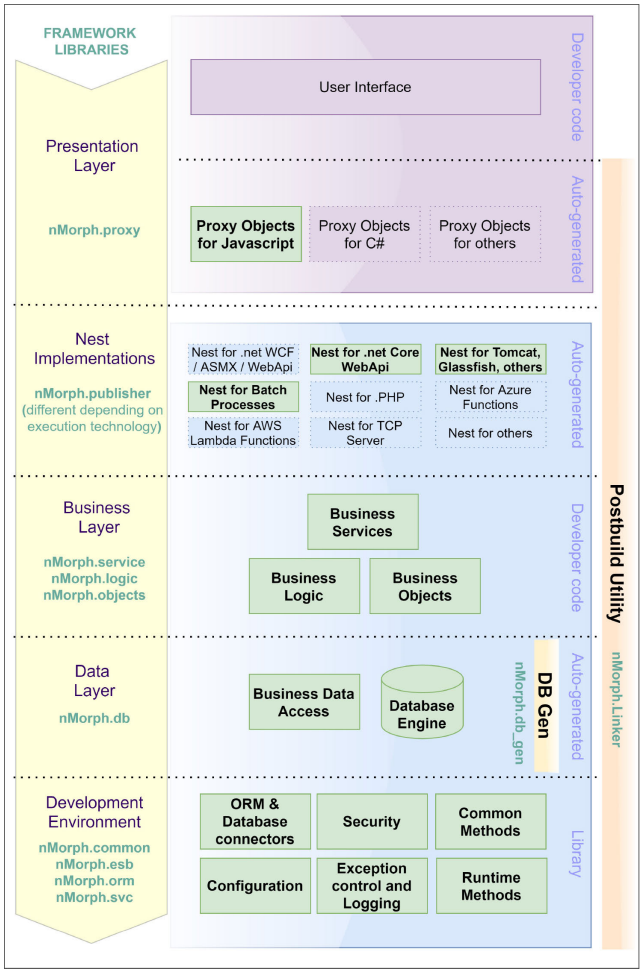


FIGURE 1. Framework libraries.

Concurrently with ORM construction, a data access layer, nMorph.db, was developed. This layer delineates objects corresponding to the database tables and views. From this model, the nMorph.db_gen tool was created to connect to a development database, discover its structures, and automatically generate objects for all tables and views, thereby laying the groundwork for business logic operations. This layer was then subjected to autogeneration.

Subsequent efforts have focused on business objects and logic projects. The former specifically aims to assemble complex structures that merge information from various table sources and are beneficial as data transport structures at the interface. Business logic defines the standard for generating methods the proxy library. This was implemented using the nMorph.objects and nMorph.logic libraries.

Finally, the Nest implementations were built by creating specific solutions in each target language and devising mechanisms to ensure compatibility of compiled artifacts in transpilation so that they can be directly executed as services. This layer also auto-generates the proxy, producing methods and objects analogous to business logic and objects, but on the user interface side, significantly facilitating their use. It should be noted from the referenced figure that not all the theoretical elements of the proposed design model have been implemented. The realized elements are highlighted in green in the chart. Non-critical components, which will be considered in future framework enhancement and experimentation studies, are omitted.

Construction of the framework demanded extensive architectural components and code generation development. The overarching concept is used to enhance developer productivity ensuring that the written code consistently adds business value and does not just fulfil structural architectural requirements. Figure 2 shows the outcomes of the project folder implementation and the corresponding components of the design model. This correspondence is vital for bridging the conceptual and theoretical aspects with the specific framework implementation elements.

Following this exercise and after several prototype development practices in the lab, the feasibility of its operation for building transactional applications was validated. This preparation lays the groundwork for the formal commencement of subsequent experimentation stages, intending that these experimentation exercises are viewed as real-world experiences, gradually refining the procedures of the constructed nMorph framework."

D. FRAMEWORK LIBRARIES DETAILS

1) nMorph.common LIBRARY

This represents the foundational library for platform operations. This library encompasses all the essential Haxe compilation elements along with classes and methods common throughout the architectural implementation. Notable features include:

- Configuration Implementation: Through reading and writing of flat files, this system ensures that the primary tool operation parameters are not hard-coded but are managed externally via files. This eliminates the need for recompilation or application changes. Users can define the unique configuration parameters as required for each component.
- UUID/GUID Support: Given that certain languages lack native support for data types, such as UUID (Universal Unique IDentifier) or GUID (Globally Unique IDentifier), a method was introduced to offer this support. The methods generate random numbers adhering to the RDC 4122 standards [15] and others, ensuring broad support across target languages.
- Exception Control Objects: These include methods that enable the definition of security-specific objects, such as logic and security exceptions. A helper function allows developers to integrate managed exceptions anywhere in the backend structure, differentiating them from platform or infrastructure-specific exceptions.
- Int64 Data Type Handler: Especially beneficial for platforms supporting Int64 natively, as it assists with conversions to floating data types.

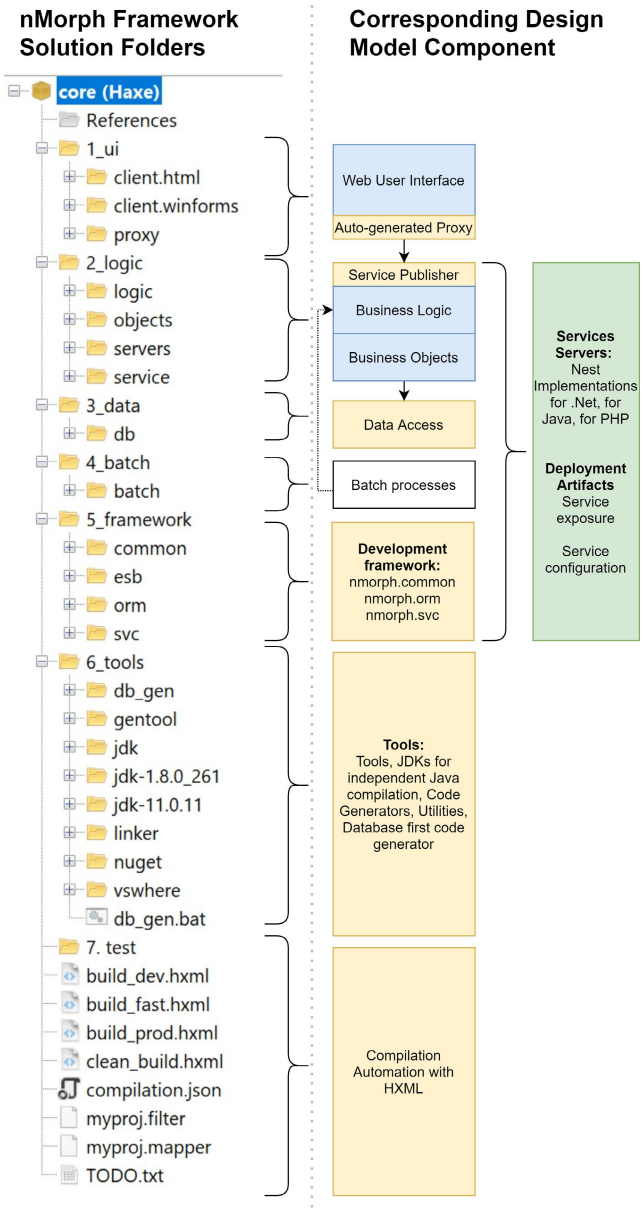


FIGURE 2. Framework folder structure.

- Serialization Methods: Utilizing Haxe's native serialization, it adopts a sequential reading format, optimizing efficiency, particularly during reading. This is particularly advantageous for processing incoming service messages and reading the configuration files.
- Interface Definitions: These can be implemented in classes across subsequent layers.

Within the postBuild program, this is the sole component where the Haxe-generated code, acting as the foundational execution library, is not cleaned. Consequently, once compiled in the target language, this library remained the only component housing execution platform element. The subsequent layers directly utilize these elements, focusing solely on their specific logic.

2) nMorph.svc LIBRARY

This library implements foundational methods useful for revealing nMorph's native services. nMorph's native methods pertain to its ability to offer web services with Haxe's inherent serialization, diverging from the predominantly used JSON or XML in most technologies. However, this does not indicate that this is the only choice. The PostBuild process also produces services compatible with other serialization methods in target technologies. However, native serialization is anticipated to perform better, enhance data transportation security, and streamline the generation of communication layers to clients in a unified manner.

Within this library, the following components are found:

- Request Object: Incorporates session and security validations and a list of logic method calls, allowing multiple logic calls in a single HTTP invocation. This boosts communication channel efficiency and execution speed.
- Response Object: A processing outcome message, an error code message (if any), and a result value list, capturing the outcome of each invoked logic method.
- Logic Method Call Object: Specifies the module, class, method, and parameters used for the dynamic method call.
- Result Information Object: Dictates the result returned by logic individually.
- Router Class: This is the core processing method for incoming HTTP requests from any target technology. This is the entry point for external calls into the internally developed Haxe logic. This method adheres to the process illustrated in Figure 3:
  - The HTTP request, in textual format within the message body, arrives and enters the router for processing.
  - The received text is deserialized, expecting a Request Object.
  - Session and security validations ensure the request validity. Invalid requests are denied by presenting a response with an error message.
  - Given the ability to make several logic method calls in one request, each is invoked accordingly.
  - The initial invocation step involves the use of reflection for locating the class that implements business logic. If the class is absent within the execution environment, an error message halts processing.
  - Subsequently, an instance of the identified object holding the business logic, is created dynamically. These objects are presumed to initially have a parameterless constructor.
  - Invocation parameters are prepared in their respective formats for dynamic entry into the business logic method.
  - Using Reflection methods for dynamic class and method searches, ensuring compatibility with all target technologies, the chosen method is invoked dynamically. Failure to locate the desired business

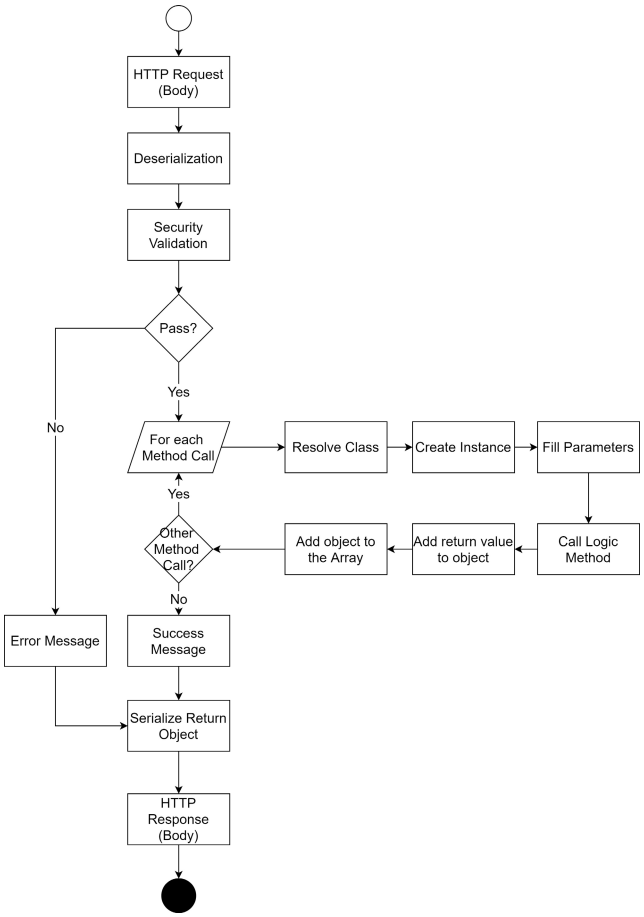


FIGURE 3. Router process.

logic method results in an error message and halted processing.

- The business logic method's return value is captured and added to an object housing each method's response.
- These methods are appended to the response array, ensuring that they are included in the result list when multiple invocations occur.
- Once all requests have been processed, an object with a success message and the results from all invoked methods are generated.
- If processing errors arise, global exception capture occurs. Exception management methods produce an error code response. Error codes differ based on the error type. If it is a business logic or security exception, it is deemed an intentional exception set by the developer, so the returned message is directly displayed to the user. For general exceptions, interpreted as infrastructure or platform errors, a log file is generated, and only the file name is shown for future reference.
- The response object undergoes serialization via Haxe's native serialization method.
- It is handed over to the target technology to be relayed to the client as a response body. It is anticipated that the client possesses the necessary methods for processing the deserialization of the response and data utilization at the user interface.

3) nMorph.orm LIBRARY

This library implements an Object-Relational Mapping (ORM) to enable the use of classes and methods for executing queries and transactions on a relational database. Non-relational databases have not yet been considered. It auto-generates SQL statements in the native language of a running database engine, incorporating transactional control, change detection, and unified recording. This simplifies the programming of the business logic and other layers within the programming structure. This library provides developers with rapid programming, versatility, and execution efficiency.

As a library implemented in Haxe, its implementation is also transpiled alongside business logic. Therefore, it can execute statements in the database equivalently across all target programming languages and supported databases. This generates a broad range of foundational technological support for selection and interchange.

The design of this tool is generic and agnostic to the database engine, which means that, by adding connectors for new database engines in the future, its operational support can be expanded without the need to alter the programming logic that utilizes it.

In Figure 4, the internal components of the implemented ORM are shown. These are detailed below, from bottom to top, with reference to the diagram.

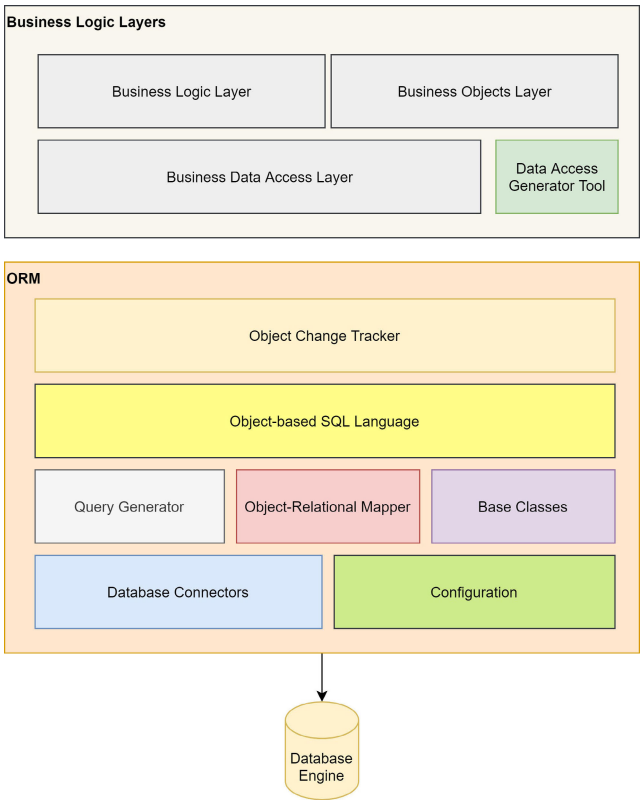


FIGURE 4. ORM components.

- Database Engine
  - The ORM design allows for the use of any relational database engine that adheres to SQL standards for connectivity to the tool and is supported within the nMorph implementation.
  - Developers will integrate their tables, views, and other elements within the database to work with them during development and runtime. The current ORM concept is based on the database-first strategy.
  - Supported databases currently include SQL Server, Oracle, PostgreSQL, and MySQL/MariaDB.
- Database Connectors
  - While the ORM's design is intended to be database engine agnostic, support for each database engine brand intended for nMorph must be added. This component provides native connection methods for each target transpilable language. Connectors aim to offer generic implementations so that ORM can connect with a specific language and execution technology.
  - This component also includes the implementation of specific SQL statements for the supported database engine, if required.
  - After the transpilation process, during the PostBuild process, the database connectors were merged with the main source code through code injection techniques. Thus, when the platform's native compilation is executed, an ORM with native connections to supported databases is available.
- Configuration
  - For database connectors to operate, appropriate parameters are required to establish a connection with the database. This component defines the essential parameters for assembling a connection string depending on the supported database engine, making it configurable as part of the general nMorph configuration strategy, based on configuration files.
  - The ORM configuration objects define parameters such as the database engine, execution environment (development, testing, and production), authentication type (credentials and Active Directory), data source, service name, SID, UserID, Password, Database Name, Port, Connection Pool Size, and other optional connection string texts.
  - Depending on the operating database engine and provided parameters, the ORM generates the corresponding connection string for a successful connection.
- Base Classes
  - Mapping objects, which are generated equivalent to the database objects, require base classes that provide them with the basic functionalities needed to operate with ORM.
  - The primary class is ModelObject, which implements variables for defining columns to map, a change controller for objects with modification tracking, generic field access for reading and writing, and a unified recording trigger.
- Object-Relational Mapper
  - When executing a query, object mapping is a key element in easing the task for the programming team of the business logic layers. Developers expect the execution of database queries to translate table and view data into objects and then use these objects to generate insert, update, or delete transactions.
  - This component is used when obtaining database results that need to be used in business logic as objects.
  - nMorph implements a scheme that allows the direct mapping of query results to a given provided object model, which is an algorithm that matches class field names with column names retrieved from the database. In this manner, any object type can be linked, and it will automatically fill or map fields that match in name, even if the object does not fully match the query structure.
  - nMorph also offers a mapping scheme based on a dynamic data management structure in memory, such as QuerySet, QueryTable, QueryColumn, and QueryRow objects. These allow information management without a defined structure and enables the structure to change dynamically as needed. This object structure is inspired by what is available in .net Framework's System.Data.DataSet library.
- Query Generator
  - The query generator is a set of internal methods spread throughout the implementation, allowing the ORM to write the final SQL code that will be sent to the database connector for execution on the connected database engine.
  - To generate the query, object-based SQL language is used, translating its composition and instructions into the specific SQL code to be executed. The generated code will vary in its writing form depending on the connected database engine and the implementation of the corresponding connector.
  - Each object used for writing the object-based SQL language has its respective SQL query generation implementation for each database through the SCRIPT_PART method.
  - In addition, developers can obtain the text corresponding to the generated SQL code to view the queries produced by the query generator.
  - The query generator can write queries either as a single or multiple queries at once. The latter is required in certain scenarios where multiple table or view results need to be accessed in the same query to facilitate object assembly.

- -- Invoking Stored Procedures does not generate a particular query but uses the database's native calling objects.
- -- Values incorporated within the query construction does not use string concatenation, to prevent SQL injection. The parameters provided as part of the native connectors are filled using query parameters.

- Object-based SQL Language
  - -- This is the visible layer for developers, through which they can use various specialized objects to assemble the custom queries required for constructing business logic layers.
  - -- This is not an independent programming language, but rather a library of objects that concatenate to form queries, as if working with SQL but through objects. Hence its designation as a language, facilitates developers understanding.
  - -- This object library's functional conception aims to allow developers to assemble queries in their programming in the same logical way they would assemble them using SQL. Hence, all query statements start using the SQL object. At the end of each query, methods are available to execute the batch and to obtain the resulting data.
  - -- Starting with the SQL object, sequential objects can be concatenated for different query statements. The main methods are SELECT, INSERT, UPDATE, DELETE. After these statements, subsequent ones can be concatenated, such as FROM, WHERE, JOIN, and many others. In Figure 5, the internal object and method definitions of the SQL object with their possible navigation and query construction paths are provided. A clause is a main language statement that serves as the backbone of the query structure. Only certain clauses can follow others. For example, the SELECT clause can only be followed by the FROM clause. Arrows in the diagram indicate the expected clause sequence.
  - -- A clause contains internal information that determines its uniqueness within the query. The following details the various objects available for this purpose:
    - * Source refers to a data source that, typically refers to the internal representation of a table or view. Within nMorph, the following sources were implemented: AliasSource, InlineViewSource, StringSource, TableSource, and ViewSource.
    - * Sentence is a method that refers to a field, a function that can be applied to a field, a value, a subquery, or an aggregate function. In nMorph, the following sentences are available: Ag, Assign, Cast, FieldAliasSentence, FieldIndexDefinitionSentence, Fn, Mt, St, SubQuerySentence, and Val. The latter allows for the incorporation of specific values into the query such that they can be passed as parameters to the native connector.
    - * A Condition is a comparison between fields primarily used for filtering, and it can be concatenated with several other conditions using the AND and OR operators.
    - * Additionally, ConditionGroup allows for the assembly of complex conditions separated by groups.
    - * FieldSentence is a specialized form of Sentence that specifically refers to the definition of a field within a table or view.
    - * Set refers to the method that allows for the assignment of a Sentence that has a value to be assigned to a FieldSentence that determines the field to which that value will be assigned. This element is commonly used with Update and Insert clauses.
    - * Field is a specialized form of Sentence that manages a data type from the database, translated into ORM language. Among these, the following are available: BigIntegerSentence, BigTextSentence, BinarySentence, BooleanSentence, CharSentence, DateTimeSentence, DecimalSentence, FloatSentence, IntegerSentence, NBigTextSentence, NCharSentence, NVarcharSentence, RealSentence, SmallIntegerSentence, UUIDSentence, VarcharSentence. These objects are mainly used for building classes equivalent to those in the database.
  - -- Once the query is assembled, various methods are available for execution. The selection of an appropriate method depends on the desired query outcome. Executing any of these methods invokes internal SQL sentence generation methods and invokes the native connector's methods to run the sentence in the connected database engine. The execution statements are as follows:
    - * SCRIPT: Used to generate the query in text format. This mode does not execute the query in the database.
    - * Execute: Executes the statement directly and returns an untyped dynamic object.
    - * Register: Registers this query within an existing Queue related to a change control context.
    - * Queue: Adds this query to an execution queue. It does not execute at this moment but when the query queue is executed together in a single transaction.
    - * Bulk: Registers the query within an existing Queue related to a bulk query execution.
    - * ReadValue: Reads a single value from the executed query. Reads the first column and cell of the response and returns it as a simple value.

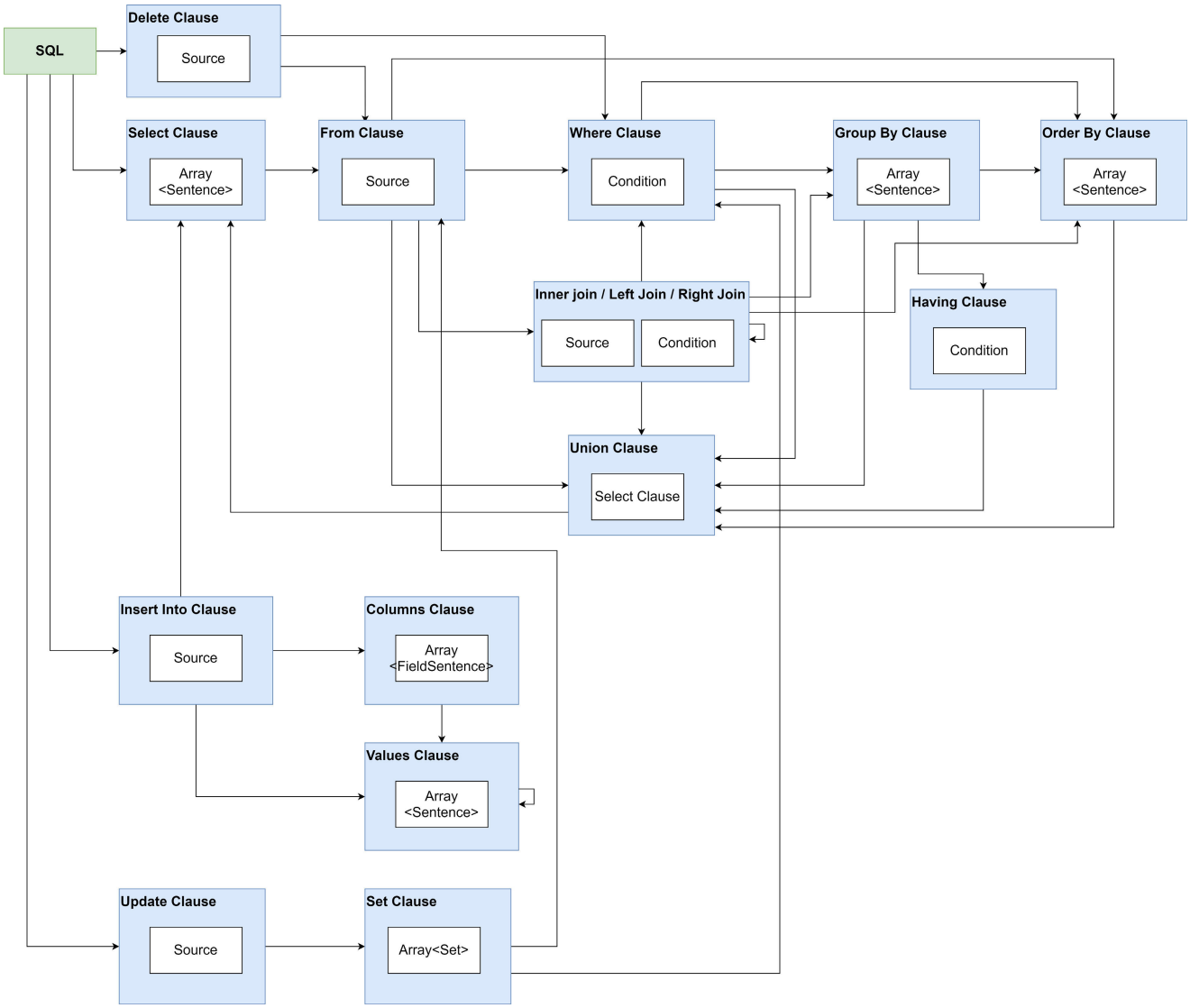


FIGURE 5. ORM query objects.

* ReadTable: Reads a table in the form of the QueryTable object, where it internally represents the information read from the query through QueryRows and QueryColumns.
* Read<T>: A generic method that executes the query and automatically transforms the query result into the data type passed as a generic parameter. It matches column names for field mapping. The passed generic object must have an empty constructor. Returns an array of objects of the generic type.
* ReadOne<T>: A generic method that executes the query and automatically transforms the query result into the data type passed as a generic parameter, similar to the previous method. In this case, it only reads one record from the query result and returns a single instance of the generic type.

- In Figure 6 section a, you can see examples of queries made with objects provided by the ORM and their corresponding queries sent to the database.

- Data Access Generator Tool

  - The object-based SQL language provides the fundamental structure for assembling queries to the database in a dynamic, flexible, and multi-platform manner, as explained in previous sections. However, its use in this form would be very extensive, minimally automated, and complex if a developer used it to develop business logic layers. Therefore, there is a need for a higher level of abstraction, built through specialized objects based on the data model of each solution. This is crucial for simplifying and encapsulating data access, reducing development time and improving code readability and comprehension for developers.

a) ORM Query examples

```haxe
// Haxe
var result = SQL
    .SELECT()
    .FROM(StringSource.SET("ExampleTable"))
    .WHERE(Condition.SET(FieldSentence.SET("Column1"), Op.Equal, Val.string("Data")))
    .Read<ResultClass>();
```

```sql
-- SQL
SELECT *
FROM ExampleTable
WHERE Column1 = 'Data'
```

```haxe
// Haxe
SQL
    .INSERT(StringSource.SET("ExampleTable"))
    .COLUMNS([FieldSentence.SET("ID"), FieldSentence.SET("Col1"), FieldSentence.SET("Col2")])
    .VALUES([Val.uuid(UUID.newUUID()), Val.string("Val1"), Val.string("Val2")])
    .Execute();
```

```sql
-- SQL
INSERT INTO ExampleTable (ID, Col1, Col2)
VALUES ('6E2604C0-FD0F-4EAC-871F-1AD5ECF5AAEF', 'Val1', 'Val2')
```

b) ORM Typed Object usage

```haxe
//Haxe
var result = SQL
    .SELECT()

    // Generated Object Table Source
    .FROM(ExampleData.TableSource)

    // Generated Object Column Definition
    .WHERE(Condition.SET(ExampleData.Column1, Op.Equal, Val.string("Data")))
    .Read<ResultClass>();
```

```sql
-- SQL
SELECT *
FROM ExampleTable
WHERE Column1 = 'Data'
```

c) ORM Tracked object usage

```haxe
// Haxe
var ctx = new Context();

var rData = SQL
    .SELECT()
    .FROM(StringSource.SET("ExampleTable"))
    .WHERE(Condition.SET(FieldSentence.SET("Column1"), Op.Equal, Val.string("Data")))
    .Read<ResultClass>(ctx); // Link to the Context

rData.Column1 = "New Value";

ctx.Save();
```

```sql
-- SQL
UPDATE ExampleTable
SET Column1 = 'New Value'
WHERE ID = '6E2604C0-FD0F-4EAC-871F-1AD5ECF5AAEF'
```

FIGURE 6. ORM example code.

- Because this abstraction layer must be built based on the data model of each solution, it cannot be constructed as an internal part of the ORM, but within the business data access layer, which is part of the business logic layers.
- Consequently, to make this concept generic and applicable to any software project, a source code generator was implemented to run at development time. This generator can connect to the database engine, and through data structure discovery statements, it writes classes equivalent to the database's tables and views. This tool follows the database-first methodology, similar to the Entity Framework. These classes implement inheritance in the base classes and use other ORM objects. They also contain methods to simplify queries and fields representing relationships, so they can be directly used for business logic. It is expected that these objects can be used for data transport, even for sending or receiving information from the application's front-end. The generated source code is in the Haxe language to ensure its transpilation capability and use in the business logic layer.
- The generation process includes the following main elements:
  - Generate a class for each table and view found in the database.
  - Generate a static instance for the Source object, referencing the table equivalent to the generated object. This will be used as a reference in FROM statements, for example.
  - Generate static instances referring to the database columns, through Sentence objects specialized in the database data types. This will serve to reference them in query statements such as SELECT or WHERE.
  - Generate quick filter methods, such as static WHERE functions, to directly incorporate filters and obtain response objects.
  - Generate direct query object retrieval methods, such as BYPK, to obtain a response record based on the primary key.
  - Integrate private fields and public access functions to represent each table column in fields with equivalent Haxe language data types.
  - Include array or reference type fields for representing tables that have relationships, so composite objects can be built based on the database's relationships.
  - Implement methods of tracking the Object Change Tracker to facilitate change detection on data and joint recording.
- Once the data access layer source code is generated, queries can be typed using the generated objects. In Fig 6 section b., it is possible to visualize an example of a query with the generated objects, along with its SQL statement.
- In the nMorph implementation, the data access generation tool is called "db_gen."

- Object Change Tracker
  - To facilitate the data manipulation process carried out by the ORM and the work with the classes generated by the Data Access Generator Tool, an object called Context has been incorporated. For all data access layer objects linked to this object, an automatic change detection mechanism is maintained.
  - In this way, once the developer feeds data into an object and links it to a context, business logic can directly and naturally manipulate the object's properties. After making all changes, the developer can invoke the context's recording method, reflecting all changes from all objects in a single transaction. All data changed in the properties of objects linked to the context will be detected, and ORM's specific statements will be automatically created to reflect these changes in the database. Finally, the context's change list is cleared for further manipulation and recording if necessary.
  - An example of context object handling is shown in Figure 6 section c, where it is possible to see the manipulation of an object and the final SQL statement that would be executed in the database engine.

4) nMorph.service LIBRARY

This library implements initialization methods for service exposure in a unified mode for all the platforms. The primary element is the reading of the configuration file, which is intended to be implemented only once and to function in all environments. The PostBuild process discovers classes containing configuration properties and sets them in this library through precompilation generation.

This library has been considered to incorporate code that makes any other required element compatible for the initialization and operation of nest implementations, in a unified way for all embedded technologies.

5) nMorph.linker TOOL

A single root file utilizing Haxe Make technology is available to run the build process for a solution based on nMorph. This is a file with a .hxml extension containing various instructions and execution parameters for transpilation. Therefore, to compile the entire nMorph project, one only needs to execute the following command: "haxe build_prod.hxml". This runs the entire process required to obtain the execution artifacts, starting from a single source code. This file calls other .hxml files, one for each layer of the nMorph solution, and within each layer is a specific call to the nMorph.linker component.

This console program implements the PostBuild Utility provided in the software architecture design model detailed in a previous study. This utility allows the code to be prepared prior to transpilation, after transpilation, executes various code preparation and cleanup procedures, compiles and generates execution artifacts, and has them ready to run on their respective target platforms, as explained later in this section. Throughout the nMorph construction process, this program is called multiple times; therefore, its process adapts based on the component calling it and its input parameters. The execution process of nMorph.linker is as follows.

- Parameter Reading and Processor Initialization
  - The parameters are passed as initialization parameters for the console program. They read parameters from the solution's directory folder, the current subproject folder, subprojects referenced from the current one, the current generation namespace, compilation environment definition (Fast = Quick Compilation, Dev = full development compilation, Prod = full production compilation), logging method definition (Console, log file, both), and rendering method definition of the current subproject (PreCompile, Publish, ORM, UI, Logic, Batch, PostBuild).
  - The program additionally reads general compilation parameters through the compilation.json file, which is typically located at the root of the nMorph solution. This file defines the major and minor version numbers with which all the solution components will be compiled. Note that the version management for compilation is provided by these parameters.
  - If the environment is Fast or Dev, the transpilation target folder is ''out''. If it is a Prod environment, the target folder is ''deploy''. Based on this, the path to the transpiled source code output folder was determined.
- Linking for .net Framework and .net Core
  - For .net 4.0, it compiles in all the environments, whereas .net Core 3.1 and .net Core 5.0 run only for the Dev and Prod environments.
  - The .net 4.0 preparation output folder is ''cslinked.'' The .net Core 3.1 preparation output folder is ''cscore31linked,'' and .net Core 5 output folder is ''csnet5linked.''
  - Regardless of their content, if these folders exist, they are deleted before starting the process.
  - All transpiled codes are transferred to each processing folder. If it is nMorph.common, the code is transferred in its entirety, including in the base libraries. If it is not in this library, the code is cleaned up and only specific business logic or specific subproject code is transferred.
  - For nMorph.common, a code injection process is performed for certain libraries that must be adapted to nMorph's execution scheme. These are serialization, deserialization, date management, and byte management libraries.
  - For nMorph.orm, a code injection process is performed by taking the specific connector code from the .net platforms for supported databases.
  - Assembly property files were added to define specific compilation versions.
  - A dynamic .net Visual Studio project was created using a .csproj extension that includes all incorporated elements and defines references to other projects.
  - A library restoration process is executed through Nuget commands, referring to the generated packages .config files in the target project.
  - A direct compilation process is executed, by invoking .net Visual Studio executable file, through parameters that execute its compilation without the need to open its interface. This ensures that the compilation produces complete final publication artifacts, and there is a usable code solution for a user who wishes to customize it directly in the destination source code, if needed. The final executable product is a .dll file extension.
- Linking for Java with JDK 8 and JDK 11
  - Java compilation occurs only for Dev and Prod environments, assuming that debugging and development execution will initially be performed in .net, and the Java solution will be produced only when a full build is desired.
  - A folder with a specific JDK version will be utilized. For example, for JDK 8, the base folder will be ''jdk8_0_261_linked,'' and for JDK 11, the base folder will be ''jdk11_0_11_linked.''
  - For Java, to ensure the availability of specific JDK versions, these JDK folders are incorporated within the nMorph solution. nMorph can continue to incorporate support for the main JDK versions as needed. The selection of JDK 8 and JDK 11 is because they are the main distinguishing versions given in the application servers of this technology.
  - Regardless of their content, if these folders exist, they are deleted before starting the process.
  - All transpiled codes are transferred to each processing folder. If it is nMorph.common, the code is transferred to its entirety, including the base libraries. If it is not in this library, the code is cleaned, and only a specific business logic or specific subproject code is transferred.
  - In Java, a source code file cannot contain more than 64KB in a single line. Although this is unlikely, during Haxe transpilation, it might happen because of the reflection data writing that this technology has to make it agnostic to the target technology. This information is written in a field called ''__rtti'',

which internally has an XML code written in a single line. To avoid potential later Java compilation problems, a code subdivision process is executed, searching for this field and dividing it into multiple lines such that no line reaches the compiler's proposed maximum size.
  - For nMorph.orm, a code injection process is performed by taking the specific connector code from the Java platforms for supported databases.
  - A compilation file called "FileList.txt" was created. It is a text file in which the names of the files that will be part of the compilation are described line by line.
  - The Java compiler is invoked, inputting the "FileList.txt" file and reference projects as parameters so that the compiler recognizes all elements for compilation. This process generates .class extension files.
  - Because subprojects are proposed as separate components, it is necessary to unify the compiled elements into a single execution unit. Therefore, only a unique file containing all the compiled elements only is generated. The final product is a .jar file extension

- PHP Publication
  - PHP is not a compiled language, but is only interpreted. This implies that it does not require a post-transpilation compilation process. Instead, specific procedures are requiered to consolidate all the source code into a single folder for execution as a unified entity.
  - All transpiled codes are moved to each execution folder. If it pertains to nMorph.common, then the entire code, including the basic libraries, is transferred. If not, the code is cleaned, and only parts related to business logic or specific subprojects are transferred.
  - At the end of the entire process, all source code is consolidated into a single executable folder, operable from the root.

- Generation of Standard Services for Each Technology
  - Although nMorph provides a native service exposure method, it is suitable for proprietary projects, but not for third-party backend integration projects. Standard technologies are necessary in these scenarios. In this step, an automated source code generation process is executed, allowing logic methods to be interpreted as operational methods for the target technologies.
  - Automatic interface exposure generations for ASMX XML Web Services, Windows Communication Foundation WCF, and WebApi REST Services. The latter is compatible with both, .net Framework and .net Core. For Java, a SOAP Web service with Spring and a JAX-RS REST API service are automatically generated. For PHP, a web service exposure based on an interpreter is generated.

- Compilation of Executable Services
  - Throughout the nMorph construction process, each the partial components of each subproject are created separately. It is essential to incorporate them directly into a single solution that represent the service call entry point that the platform will support. The corresponding nested implementation for each technology was used, which already contains integration source code and standard service codifications for each technology.
  - In the final stages of nMorph construction, an additional compilation process is executed for each technology. This compiles the nested implementation, linked libraries, and source code of the services into a single, directly executable solution.
  - For .net Framework and .net Core, a direct compilation with Visual Studio .net was performed using parameters that allow this process to run without a visual interface. The result is a solution that is ready to run services using these technologies. Separate solutions are available for each supported version. For the final deployment, Visual Studio's publishing tools or Xcopy-based publishing can be used.
  - Java-based communication relies primarily on servlets. The nesting compilation cannot be generically performed for all application servers but leverages the Servlet implementation libraries provided by each. For Tomcat 10, TomEE, Glassfish 5.1, and Glassfish 6.1, the CoreJakarta.java application server library will be integrated for use with the Jakarta libraries. For Tomcat 9, Weblogic, Websphere, and WildFly, the CoreJavax.java library was integrated for use with the JavaX libraries. As all application-server diversity must be supported in a single process, a separate folder and specific compilation for JDK 8 and JDK 11 will be created. Each compilation references the JAR libraries compiled throughout the process, the generated source code of the web services, and the base servlets library of the application server. Finally, a packaging procedure is executed, resulting in .war file, available separately for each application server, and compatible JDK. This file is finally deployed on nMorph's integrated application servers for the developer to choose their preferred deployment or working server.
  - For PHP, no additional compilation process is required because it is an interpreted language. The generated code is integrated into a single folder, which is the root of the portable publication server used for development. Thus, the code can be executed directly. The generated PHP code can be run on any PHP version above 7.0.0.

- Generation of Back-end Proxies
  - When using native services, nMorph provides a Typescript/Javascript front-end library for easy and swift back-end connection. This library consists of base and solution-specific methods.
  - An exemplary website was developed within a Visual Studio solution, embedding the generated library and reference screens. This allows the testing of the entire front-end and back-end execution chain, including database transactions and business logic.
  - Regarding base methods, nMorph uses a subproject named Proxy containing references to the Haxe libraries for native serialization and other data transformation methods, ensuring client-server consistency. This project was then transpiled to Javascript. The generated source code functions as a single functional unit; therefore, the script's container structure must be modified for use as an open library across all website screens.
  - Concerning the solution's specific methods, the generator references all the business logic objects and methods marked for exposure. In this case, Typescript classes and methods that implement their invocation are written. These libraries equip programmers to call them as internal functions and retrieve the call results for interaction with the final screens. This framework can invoke multiple logic methods in a single call.
  - After these generation processes, a Typescript compilation process for the generated folder is run, providing the equivalent Javascript version, allowing the website execution.
- Generation of Configuration Files
  - Once the code is deployed, there is a root folder for configuration files, providing a unified location for development application servers to read the execution parameters. Configuration files were serialized using native serialization methods. If these files are missing in the target folder, they are auto-generated based on the default configurations.
- Batch Process Preparation
  - Although the execution of a web application as a direct consumer of the exposed logic methods has been largely analyzed, the back-end can also be utilized via batch processes. These processes are typically used for long-durations or for asynchronous processing.
  - nMorph has a specific subproject for programming logic methods to be invoked using this methodology.
  - If targeting .net Framework or .net Core, the compilation process of the subproject changes to produce an .exe file. This program can be manually executed or scheduled in the OS using the Windows Task Scheduler or CRON in Linux, to name a few examples.
  - For Java, a JAR file is generated, linking all incorporated libraries and allowing standard command execution. This program can also be manually executed or scheduled on the OS using the Windows Task Scheduler or CRON in Linux, among other methods.
  - For PHP, no specific preparation is required, but its execution is done through direct console commands, which finds the entry point of the batch program.

It should be noted that nMorph's construction mechanism is ready to operate under Continuous Integration (CI) methodology, allowing the entire solution to be compiled with a single command, generating artifacts and deploying them to the appropriate paths for automatic availability on the chosen application server.

Currently, nMorph does not propose a mechanism for scheduled or automated testing. This aspect could be a subject for further investigation beyond the current research.

E. SOFTWARE DEVELOPMENT LIFE CYCLE PROCESS CONSIDERATIONS

The nMorph framework introduces a novel approach to designing multi-programming language software. When unveiling its internal structure, one can discern the technical prerequisites necessary for its construction and operational elements that cater primarily to back-end developers. However, it's imperative to draw a correlation between the nMorph framework and the Software Development Life Cycle (SDLC). Broadly speaking, the SDLC encompasses phases such as analysis and planning, design, development, testing and deployment, and maintenance and support.

For this examination, we've prepared the Figure 7, which delineates an input-process-output schema. Here's a detailed elucidation:

In the 'Input' section, the primary input elements are the software requirements, either from the client or from the envisioned product. To address these, specific methodologies and tools from requirements engineering will be employed. The use of the nMorph framework doesn't influence or modify these methodologies.

The 'Process' section lays out the various SDLC phases. During the analysis and planning, as well as the design phase, prototyping tools or other preferred methods and tools will be adopted by the team. The nMorph framework does not influence these phases. However, in the development, testing and deployment, and maintenance and support stages, there are variations introduced by the nMorph framework. This stems from the need to primarily switch development tools, as traditional programming languages are superseded by languages compatible with nMorph. This entails preparation of development environments, identification of suitable application servers for development, testing, and production

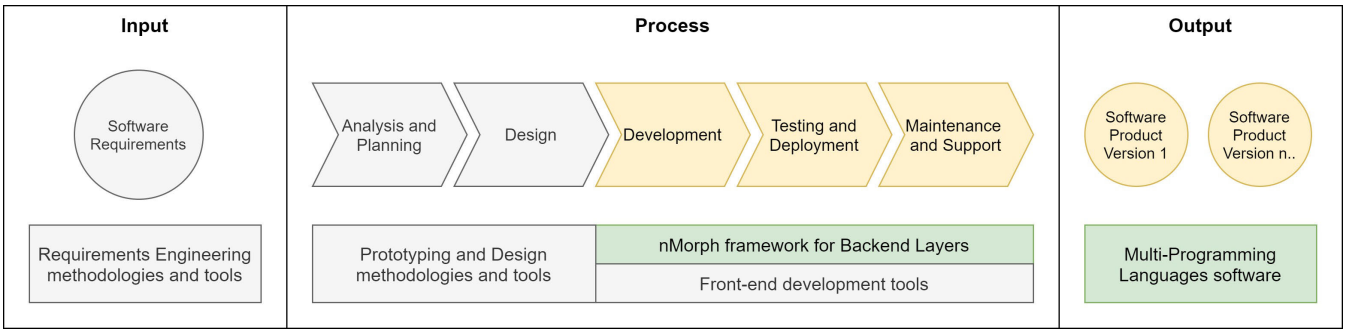


FIGURE 7. SDLC related to nMorph.

stages, utilization of integrated development environments (IDEs), code versioning tools, and the training of the software development team. This is especially relevant as most individuals may initially be unfamiliar with this emerging technology.

It's noteworthy that nMorph predominantly addresses the back-end layers of software, meaning front-end layers will be developed using the chosen programming language and framework. nMorph automatically crafts a proxy layer, aiding front-end developers in invoking logic methods exposed from the back-end. Thus, they must familiarize themselves with this to streamline their development cycle.

In the 'Output' section, with the integration of nMorph, one can not only achieve a complete and functional software but also implement it across various programming languages simultaneously, facilitating its deployment in diverse environments.

Regarding documentation implications, quality assurance, change control, project and time management, employing the nMorph framework necessitates no particular alterations.

F. FRAMEWORK REPLICATION ARTIFACTS

The source code of the framework was organized and published on GitHub (https://github.com/anfebafu-epn/nMorph) for reference to the aforementioned implementation.

III. EMPIRICAL EXPERIMENT: THROWAWAY PROTOTYPES

A. CONTEXT

The implementation of the nMorph framework covers all conceptual elements proposed in the design model [5]. However, it is worth noting that the criteria and validity level of the software architecture design proposal and framework are conditioned by the generation of usage experiences and applications, as well as the quality of their outcomes. A greater number of practical exercises and real scenarios will provide a better opportunity for improvement, thereby offering stronger support and a foundation for scientific contribution and its acceptance for widespread application.

Thus, it is neccessary to take the initial steps to establish the practical validity of the framework's application. It is essential to set up different types of application and usage exercises in controlled environments to facilitate data collection, identify and incorporate enhancements, and stabilize all components, gradually increasing the reliability of its application and thereby gaining confidence for subsequent larger-scale exercises.

Consequently, a laboratory test scenario was designed involving individuals other than from the framework development team, involving external developers. This study aimed to test the framework components by proposing a unique predefined exercise with defined functionalities related to basic bank account management, allowing the establishment of certain comparative metrics and necessary feedback. The expected result was to have information relating to their development experience, its operation, perceptions, and the general acceptability of the proposal through a survey and tabulation of practices found in the source code.

B. OBJECTIVE

Engage external programmers in a controlled testing exercise through which a throwaway prototype is implemented. Instructions would guide participants to implement business logic in widely known simple banking transactions. This implementation was to be done twice, once directly in a designated target language and once in the Haxe language, using the nMorph framework.

This dual implementation provides comparable information between the solutions. Data collection was performed through an in-depth survey and source code review, allowing for result analysis, generating a better understanding of its applicability, perceptions of the ease of use of the framework, perceptions of the development team's efficiency, and providing an opportunity for tool stabilization. The main objective is to gain a deeper understanding of the framework's application experience for programming business logic in external actors facing the newly proposed technology for the first time.

C. RESEARCH DESIGN

An in-depth survey method was chosen because it involves a controlled exercise with a defined, limited, unique, and comparable scope, based on perceptions. This allowed the involved individuals to fill out a detailed questionnaire

about their experience while using the nMorph framework. The survey facilitated collection of both quantitative and qualitative data.

In addition, the resulting source code also served for a detailed analysis of the programming practices or issues encountered by the participants during the exercise. This information helped to identify patterns to improve the stability and maturity of the tools. At the end of the exercise, the source code solutions were not used for any additional purposes, thus making them throwaway prototypes.

This design aimed to find prominent evidence from various perspectives that could generate new observations regarding the applicability of the nMorph framework. The primary focus was to answer why developers would use certain elements in their programming and explain the origin or correlation of the practices effectively applied when using new technologies.

The survey design parameters are as follows:

1) SCOPE

Development of an application with basic functions applicable to a bank account to demonstrate the applicability of implementing business logic using the nMorph framework, compared to implementing business logic in an assigned programming language.

2) PRIMARY METHOD

Quasi Experiment: The Quasi Experiment approach [16] is ideal for real-world settings, such as this research. Without requiring complete randomization or control over external factors, this method allows a comparative analysis between the nMorph framework and traditional programming languages, reflecting real-world scenarios and offering direct insights into the framework's practical applicability.

3) SAMPLE SELECTION

An open invitation was extended to the group of participants who met the target group. Nineteen junior developers were enrolled. Participants were students from the final level of the software development career. Fourteen of them completed the survey forms after finishing the exercise activities. Confidence intervals for the collected data were calculated based on the actual sample size, which is an element included in the result analysis.

4) PROCEDURE

- Recruitment of novice software programmers who participated in the exercises.
- A draw was held, assigning a different programming language to each participant, and choosing between PHP, C#, and Java.
- Each participant received specific educational material about their assigned language, in addition to specific content on the Haxe language, nMorph framework, and proposed ORM.
- Adequate time was given to participants to review the material and apply practice exercises to ensure mastery of the technologies involved.
- Complementary lectures and instructions for the main exercise to be developed were provided.
- Each participant executed the same programming exercise twice. First, in the assigned programming language and second in the Haxe language using the nMorph framework.
- A survey form was handed out to be filled out during the exercise, which was collected along with the source code.
- Procedures for processing, data analysis, and information tabulation were performed to generate conclusions.

5) DATA COLLECTION

The survey was developed in Excel. It was handed out to each participant at the start, noting the various perception variables of knowledge, understanding, and time consumption during and at the end of the development process. In addition, each participant had to submit two source code solutions: one developed in the assigned language and the other using the nMorph framework.

6) DATA ANALYSIS

Information from the Excel format was cleaned and transferred to a single Excel format consolidating information from all participants. Formulas and correlation analysis were applied to the obtained values to explain the various behaviors and factors to be observed.

7) ACTORS IN ANALYSIS

Novice software developers who participated in the exercise.

8) MAIN RESEARCH QUESTION

To what extent is the nMorph framework applicable for novice developers writing business logic?

9) SUB-QUESTIONS

- What are the main comparative parameters for determining the efficiency and effectiveness of applying the nMorph framework compared to applying a direct target language? - What programming practices are applied by programmers using the nMorph framework compared to those used in other languages? - What challenges or difficulties were presented during the study that could help improve the framework for future exercises?

D. EXECUTION

1) Recruitment of Programmer Participants
An agreement was established with a local university's software programming department to invite various students interested in participating in the project. Nineteen applications were received, and ultimately, 14 individuals participated in the exercise from start to

end. The participants' profiles corresponded to those of novice programmers.

They were students at the final level of the software programming degree. All participants had honed their skills in algorithm creation, were familiar with various programming languages, had experience with database connections, and had knowledge of several software development techniques and methodologies, intrinsic to the professional course's curriculum. For these participants, the presented project was an opportunity to acquaint themselves with new technologies and garner experiences to prepare them for professional projects in their future careers.

2) Programming Language Lottery

During the nMorph framework implementation, the target programming languages for transpilation were selected. Accordingly, it was imperative that a double implementation be executed during the exercise presented to the participants. Initially, using a direct target programming language, and then, the Haxe language by employing the nMorph framework. This dual approach allowed for a comparison of the experiences in both scenarios. A virtual meeting was held with all participants, during which a lottery was conducted, and a programming language - either PHP, C#, or Java - was randomly assigned to each of them.

3) Education on the Assigned Programming Language

Although the students had received formal education at the university about various target programming languages, it could not be guaranteed that they had comprehensive and unified knowledge of the programming language allocated to them through the lottery. As a result, an explicit education process was initiated to ensure equal footing for all and minimize any disparity due to foundational knowledge. Given that business logic typically requires a mechanism for database connections, generating queries, and transactions, this educational process also covered the use of an ORM framework specific to each target programming language to ensure comparability with the ORM proposed within the nMorph framework.

Courses for the following primary technologies were procured: For C#, the focus was on the language syntax of C# and the database connection to the SQL Server using the Entity Framework. For Java, the emphasis was on the language syntax for Java and the database connection to the SQL Server using JPA. For PHP, the participants were introduced to the language syntax for PHP and how to connect to the SQL Server database using Propel.

Specific online video courses for these selected technologies were acquired from specialized Internet platforms and provided to the participants. An 8-week period was allocated for the participants to autonomously augment their education. Upon completing this phase, a foundational knowledge evaluation was conducted using a simple exercise to ensure that the essential concepts were reasonably assimilated before initiating the project's primary exercise.

4) Education on Haxe and the nMorph framework

None of the participants had prior knowledge of the Haxe programming language. Therefore, an introductory seminar on language concept and transpilation techniques was necessary. The participants were provided with Haxe programming language-related documentation for progressive learning. They were given four weeks to supplement their education on the newly introduced technology. Subsequent sessions were dedicated to the nMorph framework and the proposed ORM, completed with live demonstrations and query resolutions. All sessions were recorded and provided to the participants for later reference.

5) Execution of the Main Exercise

After the participants completed their education phase, an explanatory session was held, and they were given specific instructions related to programming a business logic layer to implement basic bank account transaction functions. Given their familiarity with real-life bank transactions, the participants quickly grasped the logic behind bank account movements. This real-world relevance was intentionally chosen to reduce business comprehension gaps and to allow participants to focus primarily on programming practices while applying languages and frameworks. At the beginning of the exercise, participants were also provided with an Excel survey to be gradually filled out during the exercise and completed at its end. Figure 8 shows the design of the instrument used to monitor the exercise execution phase.

The exercise instructed participants to develop the same task twice. The first consisted of development in the programming language assigned by lottery, and the second in the Haxe programming language. Therefore, the measurement variables requested in the evaluation format were made twice for the two expected implementations.

Regarding the scope of the exercise, the specific development requirements were as follows. The business logic layer had to have methods for creating a bank account, registering a deposit, registering a withdrawal, and executing a bank transfer between accounts. The validation program required the creation of five bank accounts, execution of ten deposits indiscriminately in the created bank accounts, execution of ten withdrawals indiscriminately in the created bank accounts, and execution of five bank transfers between the accounts. Furthermore, there was a need to make two withdrawals with insufficient funds to verify the validations and to make the two bank transfers with insufficient funds to confirm the validations. Evidence of the transactions had to be displayed on the screen, and a procedure was

Hoja de seguimiento de desarrollo

Nombre:
Lenguaje asignado:

Aplicación con Lenguaje Asignado

	Nivel de Conocimiento del Lenguaje (1 = Nada, 10 = Experto)	Tiempo de Análisis (en minutos)	Tiempo Desarrollo (en minutos)	Tiempo Pruebas (en minutos)	Dificultad percibida (1 = fácil, 10 = difícil)	Nivel de Entendimiento (1 = no entendí nada, 10 = totalmente claro)	Observaciones o Problemas encontrados
Capa Lógica							
1. Método para crear una cuenta bancaria							
2. Método para depósito							
3. Método para retiro							
4. Método para transferencia							
Proceso Consola Prueba							
5. Crear 5 cuentas bancarias							
6. Hacer 10 depósitos indistintamente en dichas cuentas creadas							
7. Hacer 10 retiros indistintamente en dichas cuentas creadas							
8. Hacer 5 transferencias bancarias							
9. Hacer 2 retiros con saldo insuficiente							
10. Hacer 2 transferencias con saldo insuficiente							
11. Borrar todos los movimientos y todas las cuentas (para poder repetir el ejercicio)							

Aplicación con Lenguaje Multi-plataforma

	Nivel de Conocimiento del Lenguaje (1 = Nada, 10 = Experto)	Tiempo de Análisis (en minutos)	Tiempo Desarrollo (en minutos)	Tiempo Pruebas (en minutos)	Dificultad percibida (1 = fácil, 10 = difícil)	Nivel de Entendimiento (1 = no entendí nada, 10 = totalmente claro)	Observaciones o Problemas encontrados
Capa Lógica							
1. Método para crear una cuenta bancaria							
2. Método para depósito							
3. Método para retiro							
4. Método para transferencia							
Proceso Consola Prueba							
5. Crear 5 cuentas bancarias							
6. Hacer 10 depósitos indistintamente en dichas cuentas creadas							
7. Hacer 10 retiros indistintamente en dichas cuentas creadas							
8. Hacer 5 transferencias bancarias							
9. Hacer 2 retiros con saldo insuficiente							
10. Hacer 2 transferencias con saldo insuficiente							
11. Borrar todos los movimientos y todas las cuentas (para poder repetir el ejercicio)							

FIGURE 8. Survey instrument part 1.

needed to delete all transactions and database accounts so that the exercise could be repeated later.

In the survey, the following variables were evaluated for each requested development requirement:

- Level of knowledge (1 = none, 10 = expert)
- Accumulated time spent on solution analysis and design (in minutes)
- Accumulated time spent on solution development (in minutes)
- Accumulated time spent on solution testing (in minutes)
- Perceived difficulty of the requirement (1 = easy, 10 = difficult)
- Level of understanding of business logic (1 = did not understand anything, 10 = entirely clear)
- Text about observations or problems found during requirement implementation

These variables were measured separately for implementation in the assigned programming language and Haxe language using the nMorph framework.

Each participant was given access to a virtual machine deployed on a public cloud service. All participants had the same processing capabilities and development tools. Each cloud virtual machine had 4 vCores, 8GB of RAM, and 128GB solid-state drive storage. In this way, the potential gap that might occur owing to differences in processing capacity or the use of tools was minimized. This mitigated any potential impact on the execution time for each phase, which might have occurred if each student had used their own computer. A timeframe of six weeks was allotted for the exercise execution and submission of the developed tools (questionnaire and source code).

The instruments were presented in Spanish to align with the primary language of the involved novice developers.

E. DATA ANALYTICS METHODS

The primary assessment tool was a Excel file containing the survey and specific evaluation information for each participant involved in the study. After completing the exercise, we gathered 14 Excel files that provided 189 evaluation variables collected from each participant. A program was developed to execute an automated consolidation process given the volume of data contained in each filled-out format. This program reads these variables from the format completed by the participant and integrates them in a consolidated manner into a single Excel format, enabling consolidated analysis and easing the construction of evaluation matrices for each variable.

Data were evaluated directly in Excel to generate graphs and the correlation variables presented in the results. As for the collected source code, each participant submitted two folders. The first contained the source code for the implementation of the exercise in the randomly assigned programming language. The second contained the source code of the exercise's implementation in Haxe language using the nMorph framework.

F. EXPERIMENT RESULTS

Figure 9 shows the criteria used for the interpretation of the analyses conducted using the Pearson correlation coefficient.

Interpretation will be provided based on the notion that values close to 0 indicate no correlation, whereas as values approach the limits of 1 and −1, there is progressively a stronger degree of correlation. The Pearson correlation coefficient is denoted by the letter R. This correlation coefficient is used for interpreting the results when comparing between various quantitative variables collected in the study. In the graphs, the coefficient of determination, R^2, is presented to visualize how closely one variable explains the variations in relation with another variable [17].

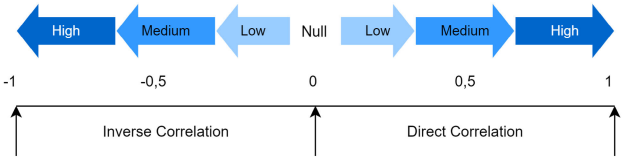


FIGURE 9. Pearson scale.

After processing and analyzing the gathered information, the established interpretations are presented in the following sections.

1) ABOUT BUSINESS LOGIC

- Measurement 1: For Measurement 1 in the Business Logic layer using the assigned language, there was a moderate positive correlation between the perceived difficulty and the time taken for analysis, suggesting that higher perceived difficulty might lead to longer analysis times. Additionally, there was a slight negative correlation with language knowledge and a low negative correlation with understanding.

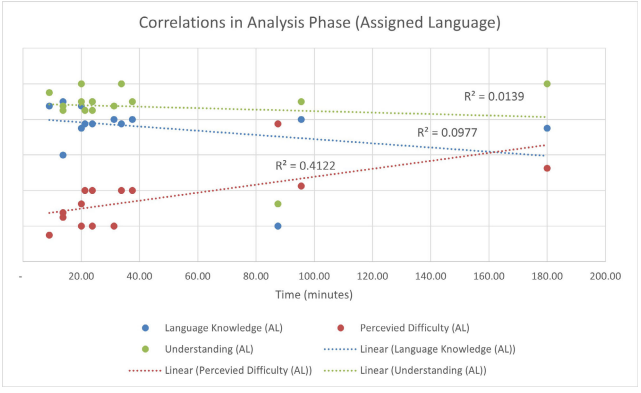


FIGURE 10. Measurement 1: Correlations in analysis phase (AL).

- Measurement 2: In the Business Logic layer using the Haxe-nMorph transpiler language, there was a slight negative correlation between language knowledge and analysis time. However, both perceived difficulty and understanding have negligible correlations with analysis time, indicating a minimal influence of these variables on the analysis duration.
- Measurement 3: In the Business Logic layer using the assigned language, during the development phase, there was a moderate positive correlation between understanding and development time. However, language knowledge and perceived difficulty showed only slight correlations, suggesting that these factors have limited impact on the duration of the development phase.

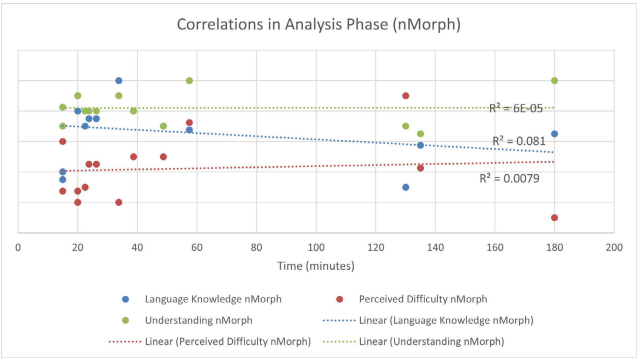


FIGURE 11. Measurement 2: Correlations in analysis phase (nMorph).

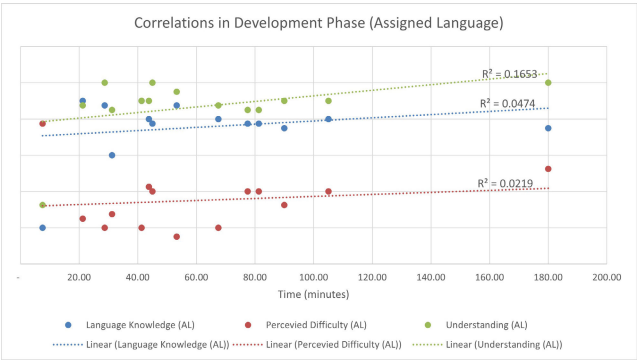


FIGURE 12. Measurement 3: Correlations in developmentt phase (AL).

- Measurement 4: In the Business Logic layer using the Haxe Transpiler with nMorph, during the development phase, there is a moderate positive correlation between understanding the task and development time. This suggests that as comprehension of the task increases, so does the time spent on development, but other factors also play a role in the duration of the development.

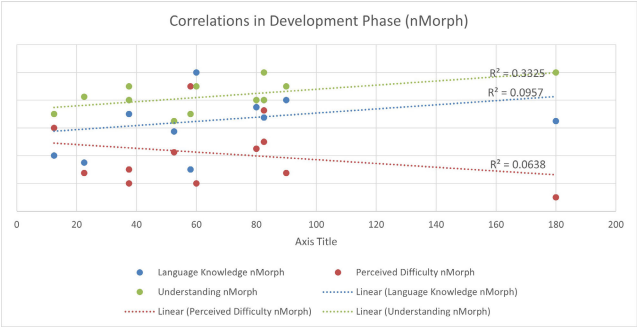


FIGURE 13. Measurement 4: Correlations in development phase (nMorph).

- Measurement 5: In the Business Logic layer using the assigned language, during the testing phase, there was a moderate positive correlation between language knowledge and task understanding with testing time. This suggests that higher scores in these areas might lead

to increased testing duration, although these correlations do not fully determine the overall dependency between the variables.

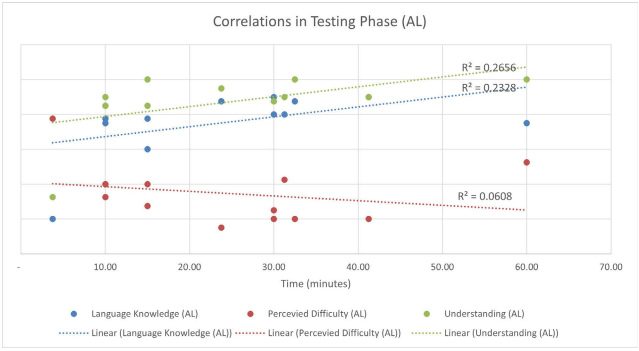


FIGURE 14. Measurement 5: Correlations in testing phase (nMorph).

- Measurement 6: In the Business Logic layer using the Haxe - nMorph transpiler during the testing phase, there was a slight positive correlation between task understanding and testing time. However, the correlations between language knowledge and perceived difficulty were negligible, indicating no significant relationship with the testing duration.

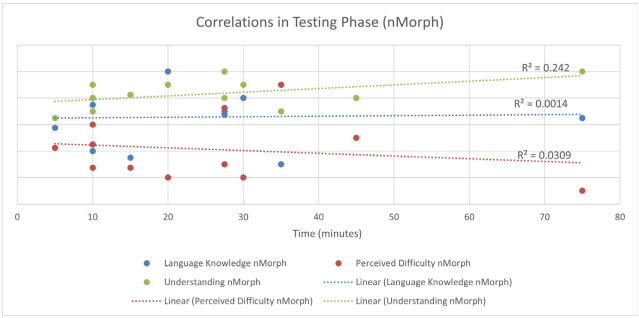


FIGURE 15. Measurement 6: Correlations in testing phase (nMorph).

- Measurement 7: In the Business Logic layer using the assigned language for the entire development cycle, there was a moderate positive correlation between perceived difficulty and the overall time spent across all phases. This suggests that, as participants found tasks more challenging, the overall development time increased slightly. However, the relationship is not strong enough to fully explain the variance in development time.
- Measurement 8: In the Business Logic layer using the Haxe - nMorph transpiler for the entire development cycle, there is a slight positive correlation between the understanding of the exercise and the overall time spent across all phases. However, this relationship is not substantial enough to indicate a clear dependency between these factors.
- Measurement 9: The Business Logic layer, when comparing user ratings for Assigned Language and nMorph, the data suggests some interesting correlations between understanding, perceived difficulty, and

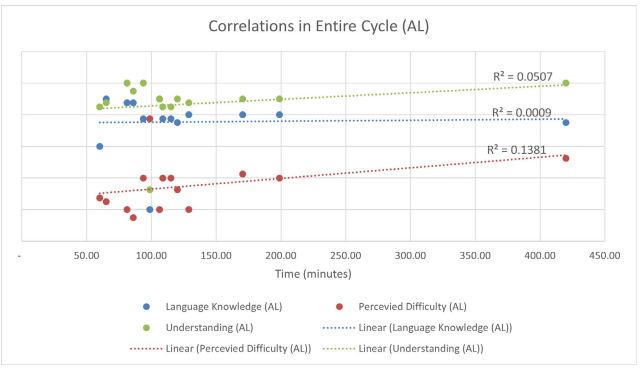


FIGURE 16. Measurement 7: Correlations in entire cycle (nMorph).

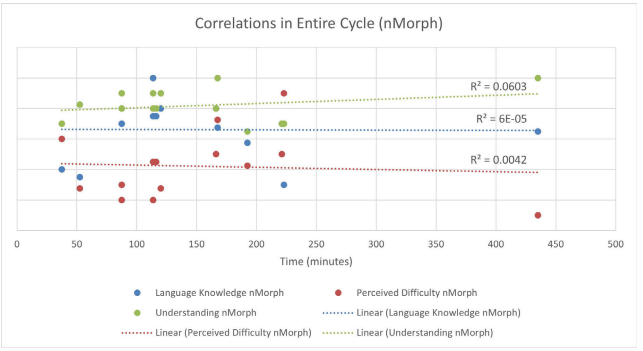
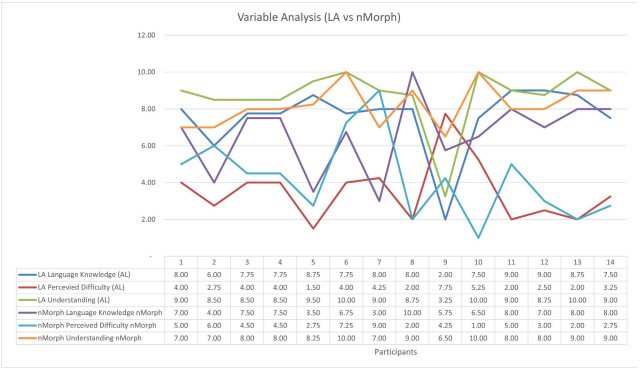


FIGURE 17. Measurement 8: Correlations in entire cycle (nMorph).

language knowledge. There was a strong negative correlation between knowledge of the assigned language and its perceived difficulty, suggesting that as participants were more knowledgeable, they found the tasks less challenging. Similarly, increased language knowledge correlates positively with understanding the exercise, reinforcing the idea that expertise and comprehension go hand in hand. Additionally, there is a noteworthy positive correlation between understanding the exercise in the assigned language and understanding it in Haxe - nMorph, indicating a consistent level of comprehension across both platforms for most participants. The other variables display moderate or low correlations, indicating that they were relatively independent or only slightly dependent on one another.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
LA Language Knowledge (AL)	8.00	6.00	7.75	7.75	8.75	7.75	8.00	8.00	2.00	7.50	9.00	9.00	8.75	7.50
LA Perceived Difficulty (AL)	4.00	2.75	4.00	4.00	1.50	4.00	4.25	2.00	7.75	5.25	2.00	2.50	2.00	3.25
LA Understanding (AL)	9.00	8.50	8.50	8.50	9.50	10.00	9.00	8.75	3.25	10.00	9.00	8.75	10.00	9.00
nMorph Language Knowledge nMorph	7.00	4.00	7.50	7.50	3.50	6.75	3.00	10.00	5.75	6.50	8.00	7.00	8.00	8.00
nMorph Perceived Difficulty nMorph	5.00	6.00	4.50	4.50	2.75	7.25	9.00	2.00	4.25	1.00	5.00	3.00	2.00	2.75
nMorph Understanding nMorph	7.00	7.00	8.00	8.00	8.25	10.00	7.00	9.00	6.50	10.00	8.00	8.00	9.00	9.00

FIGURE 18. Measurement 9: Variable analysis.

Correlation coeficients between variables		Asigned			nMorph		
		Language Knowledge	Perceived Difficulty	Understanding	Language Knowledge	Perceived Difficulty	Understanding
Asigned	Language Knowledge	N/A	-0.79472	0.88543	0.21605	-0.10923	0.43894
	Perceived Difficulty	-0.79472	N/A	-0.67968	-0.17324	0.19096	-0.26256
	Understanding	0.88543	-0.67968	N/A	0.10128	-0.07703	0.62345
nMorph	Language Knowledge	0.21605	-0.17324	0.10128	N/A	-0.49994	0.45040
	Perceived Difficulty	-0.10923	0.19096	-0.07703	-0.49994	N/A	-0.44637
	Understanding	0.43894	-0.26256	0.62345	0.45040	-0.44637	N/A

FIGURE 19. Measurement 9: Correlations.

- Measurement 10: Focusing on the Business Logic layer using the Assigned Language, the data highlights how participants allocated their time across different phases of the prototype's implementation. Specifically, the participants spent 33% of their time in the analysis phase, a significant 47% in development, and 20% in testing. Although the software development phase consumed the most time, it is noteworthy that the combined time spent on analysis and testing was substantial, emphasizing the thoroughness applied in both the preparatory and validation stages when using the assigned language.

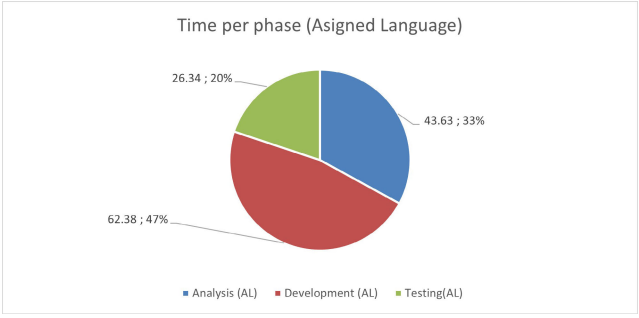


FIGURE 20. Measurement 10: Time per phase (assigned language).

- Measurement 11: Focusing on the Business Logic layer with the Haxe transpiler language - nMorph, the participants' time allocation across the prototype implementation phases is evident. They dedicated 36% of their time to the analysis, 46% to software development, and 18% to testing. Despite the use of the Haxe transpiler language, the time proportions remain strikingly similar to those observed in the assigned language. This underscores that the choice of language, whether assigned or Haxe, did not result in any significant variations in the time spent across each phase.
- Measurement 12: Centered on the Business Logic layer and comparing languages, a distinct difference emerged in the time consumption of the analysis phase between the assigned language and Haxe. The data revealed that participants generally took longer to use Haxe, with an average additional duration of 11.11 minutes. A review of individual participants' times reinforces this observation, as the majority showed increased durations with Haxe compared to the assigned language. This indicates a potentially steeper learning or adaptation curve with Haxe during the analysis phase."

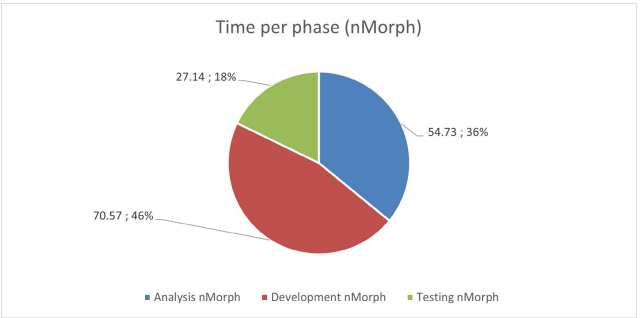


FIGURE 21. Measurement 11: Time per phase (nMorph).

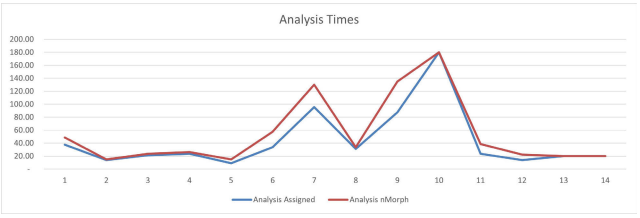


FIGURE 22. Measurement 12: Analysis times.

- Measurement 13: Focusing on the Business Logic layer and comparing between different languages, a noticeable time difference is observed in the development phase between the assigned language and Haxe. On average, the participants took longer with Haxe, with an added duration of 8.20 minutes. A closer look at individual timings reveals that while most participants required more time with Haxe, three instances showed a decrease in time with Haxe. This suggests that while many participants might have experienced challenges adapting to Haxe, a select few found it more efficient or compatible with their development approach.

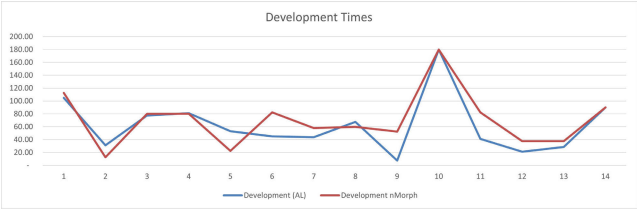


FIGURE 23. Measurement 13: Development times.

- Measurement 14: When examining the Business Logic layer and comparing the different languages, the time taken by participants in the testing phase was analyzed. Based on the data, the time distribution for testing between the assigned language and Haxe appears to be closely aligned. On an average scale, there is a negligible time difference of merely 0.80 minutes. This indicates that the transition from the assigned language to Haxe does not significantly impact time efficiency in the testing phase, suggesting that both languages offer a relatively consistent testing experience for participants.
- Measurement 15: During the Business Logic layer phase and comparing languages, the accumulated time taken

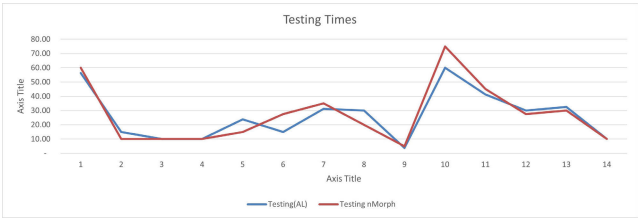


FIGURE 24. Measurement 14: Testing times.

across all tasks – Analysis, Development, and Testing – shows a discernible pattern. On average, Haxe required 20.11 additional minutes (a 15.19% increase) compared to the assigned language. There were three instances in which Haxe required less time.

Examining the specific phases, Haxe needed an extra 11.11 minutes in Analysis (a 25.46% increase), 8.20 minutes in Development (a 13.14% increase), and only 0.8 minutes in Testing (a 3.05% increase). Additionally, the perceived difficulty of using Haxe increased by 0.7 points (or 19.80%), while its perceived understanding reduced by 0.5 points (or 5.75

In essence, while Haxe might be marginally more time-consuming across most phases and perceived slightly more challenging, there is a slight decrement in understanding when participants use Haxe over the assigned language.

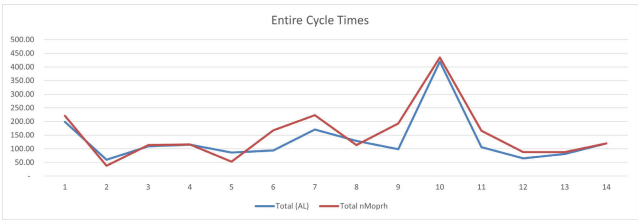


FIGURE 25. Measurement 15: Entire cycle times.

- Measurement 16: This analyzes the Business Logic layer, compares the Assigned Language (AL) and Haxe - nMorph, and computes the correlation coefficients between time spent in different phases. The findings include: a) A high correlation coefficient between the analysis time for both the Assigned Language and nMorph. b) Strong correlation between development and testing times between the two languages. Examining the specific coefficients for Assigned Language: The correlation between Analysis and Development is 0.551964, Analysis and Testing is 0.441132, and Development and Testing is 0.535227. For nMorph: The correlation between Analysis and Development is 0.559605, that between Analysis and Testing is 0.495925, and that between Development and Testing is 0.705076. Comparing across the languages: a) The correlation between the Analysis times of AL and nMorph is strikingly high 0.965148. b) For Development, it was 0.872661, and for Testing, it is 0.949311. Most of the other coefficients in the analysis are of medium magnitude. This suggests that there is a consistency in the data entry by participants, and it indicates a strong alignment in the time taken across similar phases for both Assigned Language and nMorph.

Correlation coeficients between phases		Assigned Language			nMorph		
		Analysis	Development	Testing	Analysis	Development	Testing
Asigned	Analysis	N/A	0.551964	0.441132	0.965148	0.670860	0.586436
	Development	0.551964	N/A	0.535227	0.361429	0.872661	0.603424
	Testing	0.441132	0.535227	N/A	0.333433	0.539227	0.949311
nMorph	Analysis	0.965148	0.361429	0.333433	N/A	0.559605	0.495925
	Development	0.670860	0.872661	0.539227	0.559605	N/A	0.705076
	Testing	0.586436	0.603424	0.949311	0.495925	0.705076	N/A

FIGURE 26. Measurement 16: Correlation between phases.

2) ABOUT USER INTERFACE

While the primary focus of the study pertains to business logic and the back-end layers, one can observe certain indicators associated with the implementation of the user interface originating from the new procedures and complexity of nMorph for developers. These indicators facilitate a comparative understanding of the analyses conducted on business logic.

- Measurement 17: On the User Interface layer, the assigned programming language is examined as a collective whole during the analysis phase. No discernible correlation was observed between the examined variables. The table data further supports this observation: The correlation between 'Analysis' and 'Language Knowledge' as well as 'Understanding' is slight, while there is virtually no correlation with 'Perceived Difficulty.' This suggests that for this phase, knowledge of the programming language and understanding have a minimal relationship with the time taken for analysis, and perceived difficulty does not influence it at all.

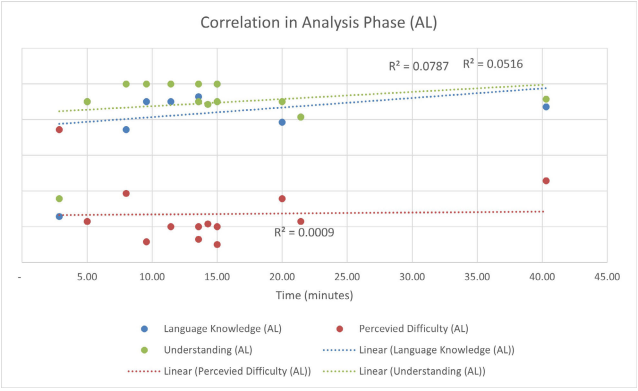


FIGURE 27. Measurement 17: Correlation in analysis phase (AL).

- Measurement 18: It focuses on the User Interface layer using the transpiler language Haxe - nMorph, the data shows a moderate correlation between 'Perceived Difficulty' and the time taken for analysis. This means that as the perceived difficulty of the task increases, the analysis time may also increase. Additionally, the provided table data reveals a slight negative correlation between

'Analysis' time and both 'Language Knowledge' and 'Understanding.' This suggests that as one's knowledge or understanding improves, analysis time may decrease slightly. However, the central determinant of analysis time seems to be the perceived difficulty.

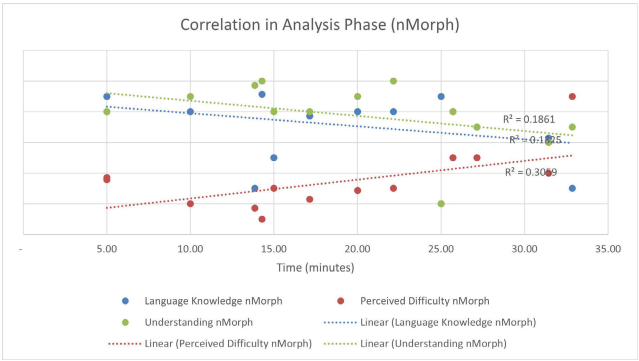


FIGURE 28. Measurement 18: Correlation in analysis phase (nMorph).

- Measurement 19: Focusing on the User Interface layer using the assigned language for the development phase, the data reveals no significant correlation between the analyzed variables. Specifically, the correlations between 'Development' time and 'Language Knowledge', 'Perceived Difficulty', and 'Understanding' are all negligible. This indicates that these variables may not be influential in the time taken for the development phase in this context.

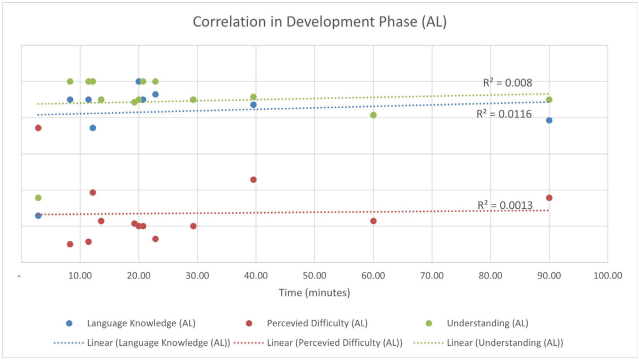


FIGURE 29. Measurement 19: Correlation in development phase (AL).

- Measurement 20: Focusing on the User Interface layer with the transpiler language Haxe - nMorph during the development phase, there is a slight correlation with the 'Perceived Difficulty' variable. However, this correlation coefficient is not substantial enough to conclusively depict the dependency between the variables. Specifically, the correlations between 'Development' time and 'Language Knowledge' and 'Understanding' are negligible, while the correlation with 'Perceived Difficulty' is low.
- Measurement 21: In the User Interface layer using the assigned language during the testing phase, A slight correlation was observed for the variables 'Language Knowledge' and 'Exercise Understanding.' However,

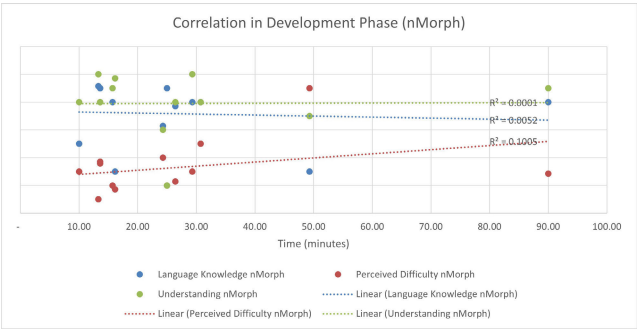


FIGURE 30. Measurement 20: Correlation in development phase (nMorph).

the presented correlation coefficients were not sufficiently robust to fully explain the complete dependency between these variables. Specifically, while the correlations with 'Testing' time for both 'Language Knowledge' and 'Exercise Understanding' are slight, the correlation with 'Perceived Difficulty' is low.

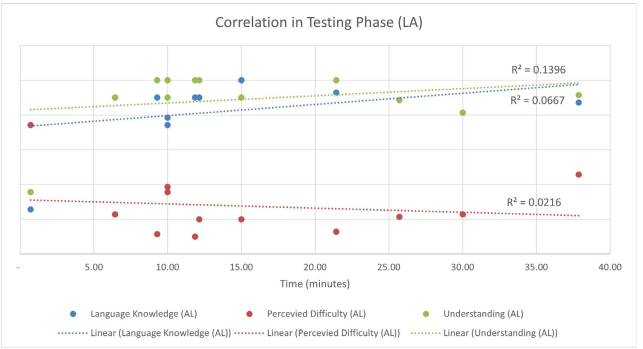


FIGURE 31. Measurement 21: Correlation in testing phase (AL).

- Measurement 22: In the User Interface layer using the Haxe - nMorph transpiler language during the testing phase, a moderate correlation was observed with the 'Perceived Difficulty' variable. This suggests that as perceived difficulty increases, the time taken for testing may also increase. Conversely, the correlations with 'Testing' time for both 'Language Knowledge' and 'Exercise Understanding' are slight, indicating a weaker relationship between these variables and testing time.

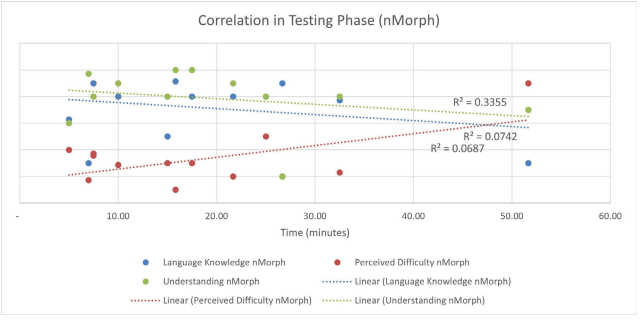


FIGURE 32. Measurement 22: Correlation in testing phase (nMorph).

- Measurement 23: For the entire development cycle of the User Interface layer using the assigned language,

a slight correlation was observed between the total time spent across all three phases (analysis, development, and testing) and 'Language Knowledge'. However, this correlation was not substantial enough to fully explain the dependency between variables. Meanwhile, there was virtually no correlation with 'Perceived Difficulty', and a low correlation with 'Exercise Understanding', implying that these factors have a minimal impact on the total time spent during the development cycle.

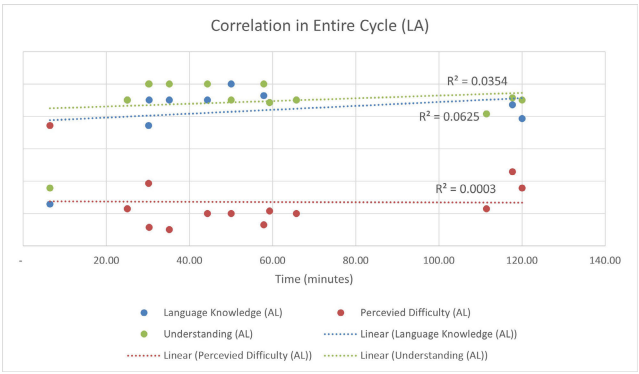


FIGURE 33. Measurement 23: Correlation in entire cycle (AL).

- Measurement 24: Throughout the entire development cycle of the User Interface layer using the Haxe - nMorph transpiler language, there is a moderate correlation between the cumulative time for all three phases (analysis, development, and testing) and 'Perceived Difficulty'. This suggests that, as the perceived complexity increases, the total execution time for the entire cycle is likely to extend. Conversely, there is only a slight correlation between 'Language Knowledge' and 'Exercise Understanding', indicating that these variables have less influence on the overall time taken for the development process.

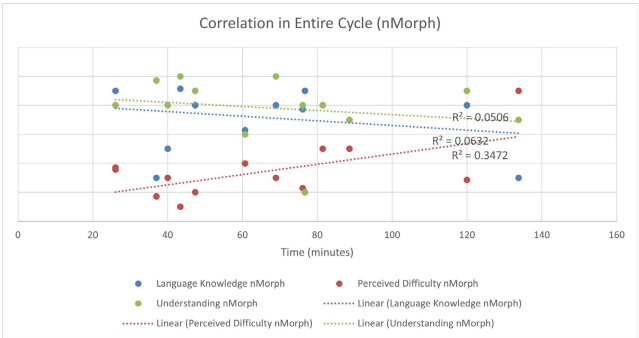


FIGURE 34. Measurement 24: Correlation in entire cycle (nMorph).

- Measurement 25: When comparing variables between the Assigned Language (AL) and nMorph in the User Interface layer, participants provided scores out of 10 for their knowledge of the language, perceived difficulty, and exercise understanding. Analyses of correlations revealed a strong negative correlation between 'Perceived Difficulty' and 'Language Knowledge' for AL, suggesting that as participants' language knowledge increased, their perceived difficulty decreased. A strong positive correlation exists between 'Understanding of the Exercise' and 'Language Knowledge' for AL, indicating that better language proficiency leads to a better understanding of the tasks. A strong negative correlation was observed between 'Perceived Difficulty' and 'Understanding' for AL, meaning that as participants' understanding improved, they found the tasks less challenging. For nMorph, none of the correlation coefficients were particularly high, implying that the variables for this language were either independent or slightly dependent. Notably, the variables analyzed for the Haxe - nMorph language did not show any significant dependence.

FIGURE 35. Measurement 25: Variable analysis.

Correlation coeficients between variables		Asigned			nMorph		
		Language Knowledge	Perceived Difficulty	Understanding	Language Knowledge	Perceived Difficulty	Understanding
Asigned	Language Knowledge	N/A	-0.87542	0.88186	0.17742	-0.02820	0.33671
	Perceived Difficulty	-0.87542	N/A	-0.80314	-0.33254	0.45781	-0.35660
	Understanding	0.88186	-0.80314	N/A	-0.00380	-0.17272	0.51077
nMorph	Language Knowledge	0.17742	-0.33254	-0.00380	N/A	-0.44039	-0.08573
	Perceived Difficulty	-0.02820	0.45781	-0.17272	-0.44039	N/A	-0.19546
	Understanding	0.33671	-0.35660	0.51077	-0.08573	-0.19546	N/A

FIGURE 36. Measurement 25: Correlation between variables (nMorph).

- Measurement 26: In the User Interface layer, using the Assigned Language for prototype implementation, participants' time allocation across different phases was examined. The data revealed that the most time-consuming phase was software development, accounting for 47% (25.97 min) of the entire process. The analysis and testing phases accounted for 25% (13.93 min) and 28% (15.69 min) respectively, indicating that combined, they exceeded the time spent on actual development. This suggests an emphasis on thorough planning and verification in the process, with the development phase consuming less than half of the total time.
- Measurement 27: In the User Interface layer, using the Haxe Transpiler – nMorph for prototype implementation, the time distribution across different phases was analyzed. Participants spent 42% of their time (28.12 minutes) on software development, while the

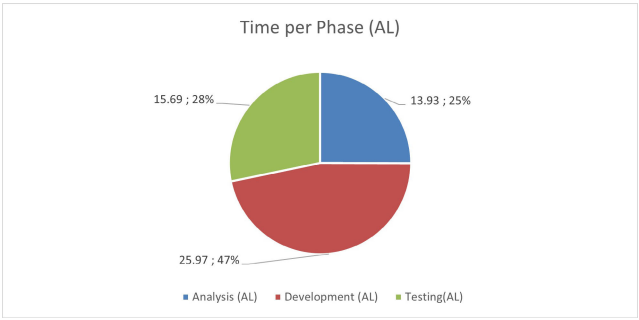


FIGURE 37. Measurement 26: Time per phase (AL).

analysis and testing phases each consumed 29% of the time, equating to 18.90 and 19.13 minutes respectively. Interestingly, these proportions closely aligned with the time distributions observed when using the Assigned Language. This suggests that regardless of the programming language–be it the Assigned Language or Haxe–the time commitment for each phase remains consistent, with no significant variation in phase-wise time allocation due to the choice of language.

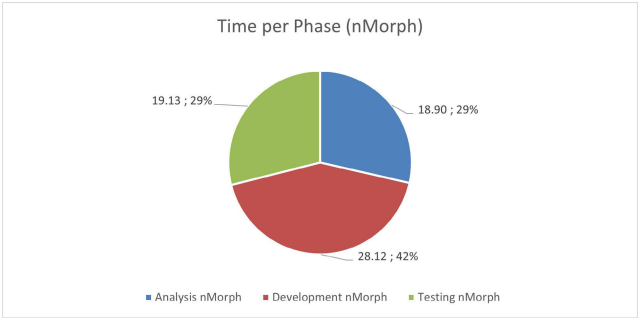


FIGURE 38. Measurement 27: Time per phase (nMorph).

- Measurement 28: A comparative study was conducted in the User Interface layer to understand the time taken by participants during the Analysis phase using two different languages: the Assigned Language and the Haxe language. The data revealed that participants typically spent slightly more time analyzing using Haxe as opposed to Assigned Language. On average, the difference in time commitment between the two languages amounted to approximately 4.97 minutes more with Haxe. This indicates that, during the Analysis phase, participants found the Haxe language to be marginally more time consuming than the Assigned Language.

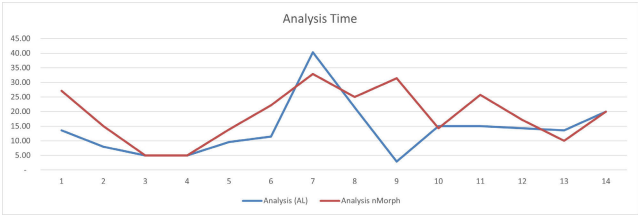


FIGURE 39. Measurement 28: Analysis time.

- Measurement 29: Within the User Interface layer, a comparison was undertaken to assess the time consumed by participants in the Development phase using both the Assigned Language and the Haxe language. The findings suggest that the Haxe language required a slightly longer duration for development tasks when compared to the Assigned Language. On an average scale, the time difference observed amounted to an additional 2.15 minutes when using Haxe. This suggests that for the Development phase, participants encountered a modest increase in time commitment with the Haxe language compared to the Assigned Language.

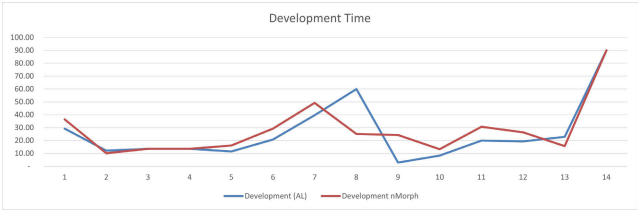


FIGURE 40. Measurement 29: Development time.

- Measurement 30: In the User Interface layer, an analysis was conducted to determine the time required by participants during the Testing phase using both the Assigned Language and the Haxe language. The data reveal that the Haxe language, on average, necessitated a marginally extended duration for testing tasks, about 3.44 minutes more, when juxtaposed with the Assigned Language. This indicates that during the Testing phase, participants experienced a slight increase in the time spent using the Haxe language compared to the Assigned Language.

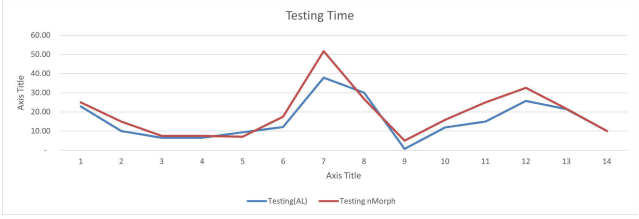


FIGURE 41. Measurement 30: Testing time.

- Measurement 31: Analysis of User Interface implementation time across different Programming Languages.
  In this evaluation, a thorough comparison was conducted between an assigned programming language and nMorph in the context of their performance across various development phases. The analysis was primarily centered on the full cycle time, comprising the stages of analysis, development, and testing. The unit of measurement was the time each participant took to complete the entire cycle.
  When observing the cycle's entirety, it was discerned that using nMorph leads to a slightly prolonged duration compared to the assigned language. On an average, this

disparity amounted to 10.56 minutes. Delving deeper into the individual stages.

1. In the analysis phase, nMorph exhibited a 35.68% increase in time, equivalent to 4.97 minutes. 2. During the development phase, there was an 8.29% increase, translating to an additional 2.15 minutes. 3. In the testing phase, a 21.90% increase was observed, which was approximately 3.44 minutes.

Collectively, the entire cycle time showed an elevation of 18.99%, equaling 10.56 minutes when using the nMorph framework. In addition to the time metrics, other parameters were also gauged: 1. There was a discerned increase in perceived difficulty of 27.17%, equating to a change of 0.73 points. 2. Conversely, understanding decreased by 12.03%, which is a reduction of 1.07 points.

These observations suggest that while using nMorph might be slightly more time-intensive and perceived as slightly more challenging, it also results in a slight reduction in the understanding of the exercise as compared to the assigned language.

Further dissection of the data revealed a general trend of increased implementation time with nMorph compared to the assigned language. Notably, time variance is more pronounced in the User Interface than in Business Logic. This could potentially be attributed to nMorph's primary focus on business logic. To elaborate:

1) For User Interface: Analysis: 35.68% increase (4.97 min); Development: 8.29% increase (2.15 min); Testing: 21.90% increase (3.44 min); Full cycle time: 18.02% increase (10.02 min); Perceived Difficulty: 55.85% increase (1.51 points); Understanding: 7.99% decrease (-0.71 points).

2) For Business Logic: Analysis: 25.46% increase (11.11 min); Development: 13.14% increase (8.20 min); Testing: 3.05% increase (0.8 min); Full cycle time: 15.19% increase (20.11 min); Perceived Difficulty: 19.80% increase (0.7 points); Understanding: 5.75% decrease (-0.5 points).

Furthermore, the data emphasized that regardless of the programming language used, business logic consistently required more time than the user interface. For instance, with the assigned language, the differences in time taken between business logic and the user interface were 29.70, 36.41, and 10.65 minutes for analysis, development, and testing respectively. Similar trends were observed for nMorph, with time differences of 35.83, 42.45, and 8.01 minutes respectively.

Finally, when examining individual participants' performance in the realm of business logic, it was evident that there were variations in the time taken between the assigned language and nMorph. For instance, Participant 1 took 65.71 minutes with the assigned language but required 88.57 minutes with nMorph. These discrepancies among the participants could be attributed to their individual proficiencies, familiarity with the languages, or other personal factors.

In summary, while nMorph demonstrated a somewhat extended implementation time across all stages when juxtaposed against the assigned language, its primary inclination towards business logic could justify this observation. Furthermore, individual variations suggest that developers' experiences and comfort with the tools at hand can significantly influence outcomes.

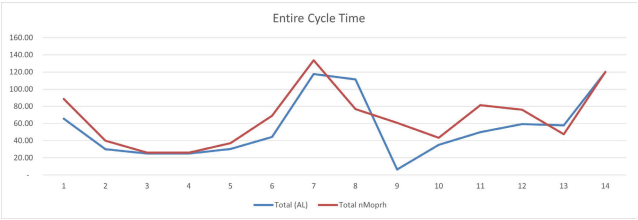


FIGURE 42. Measurement 31: Entire cycle time.

- Measurement 32: Correlation Analysis between nMorph and Assigned Language Across Development Phases in User Interface Implementation.

  This study sought to understand the relationships between the time taken during different developmental phases by comparing an assigned language (AL) and nMorph. Using the correlation coefficient as a metric, the following observations were:

  1) A high correlation was observed between the testing time using nMorph and the analysis time using AL, as evidenced by a coefficient of 0.848329. This suggests that intricacies during the analysis phase in AL might have a mirrored impact during the testing phase when using nMorph.

  2) There was a significant correlation between the testing times of nMorph and AL, standing at 0.937851. This indicates that the testing behaviors were consistent between these two languages.

  3) The development time in nMorph and AL also correlated strongly with a coefficient of 0.840780, hinting at analogous development processes irrespective of language.

  4) The testing and analysis phases within AL correlated notably with a coefficient of 0.832958, implying that the challenges faced during AL's analysis might directly influence its testing phase.

  Although these high coefficients pinpoint areas of particular interest, many other coefficients in the analysis occupy a medium magnitude. This underscores the fact that, in several phases, the time factors for AL and nMorph might not be as strongly intertwined. Nevertheless, the presence of multiple high correlation values emphasizes a considerable degree of alignment between the phases of both the nMorph and AL.

  In summary, Measurement 32 illuminated the interconnected dynamics between nMorph and AL during different developmental phases, suggesting areas of

synergy and potential opportunities for streamlined processes.

Correlation coeficients between phases		Assigned Language			nMorph		
		Analysis	Development	Testing	Analysis	Development	Testing
Asigned	Analysis	N/A	0.566697	0.832958	0.522543	0.536282	0.848329
	Development	0.566697	N/A	0.378838	0.271353	0.840780	0.236939
	Testing	0.832958	0.378838	N/A	0.417179	0.213071	0.937851
nMorph	Analysis	0.522543	0.271353	0.417179	N/A	0.454041	0.516707
	Development	0.536282	0.840780	0.213071	0.454041	N/A	0.223429
	Testing	0.848329	0.236939	0.937851	0.516707	0.223429	N/A

FIGURE 43. Measurement 32: Correlation between phases.

3) ABOUT C# ASSIGNMENT

Additionally, it is feasible to conduct specific comparative analyses for participants assigned to C#, in comparison with nMorph. This was primarily to determine whether the use of a particular target technology and its learning curve facilitated or hindered the adoption of nMorph.

- Measurement 33: The analysis of the Business Logic layer using the C# programming language breaks down the proportion of time spent in different phases. Specifically, with C#, 36% of the time was dedicated to the analysis, representing 65.65 hours. The development phase took the largest share with 45% of the time, which is equivalent to 81.65 hours. Testing, on the other hand, constituted 19% of the time, amounting to 33.25 hours. These data provide insight into how resources and efforts are distributed when constructing software using C# in the Business Logic layer.

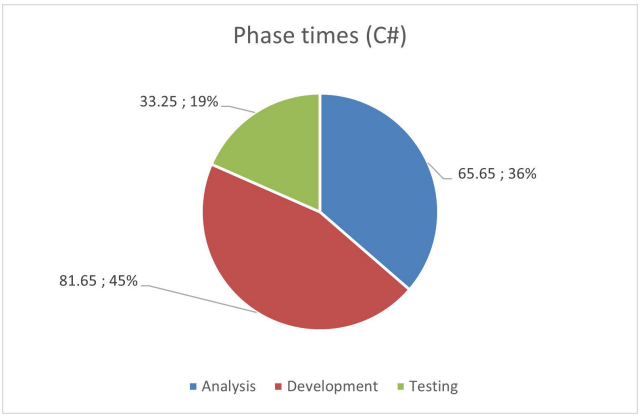


FIGURE 44. Measurement 33: Phase times (C#).

- Measurement 34: For the Business Logic layer, individuals initially assigned to C# used the nMorph programming language to measure the distribution of time across various phases. Specifically, with nMorph, 39% of the time, translating to 76.75 hours, was allocated to analysis. Development accounted for the largest portion at 43% of the time, equivalent to 86.60 hours. Testing required 18% of the time, amounting to 36.00 hours. These data suggest a relatively even distribution of effort across phases when utilizing nMorph for those initially familiar with C#.

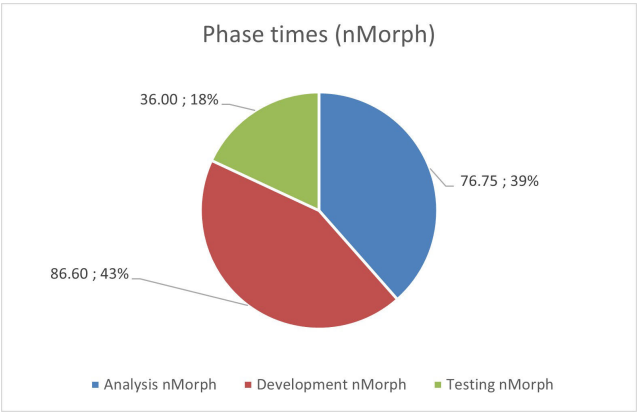


FIGURE 45. Measurement 34: Phase times (nMorph).

- Measurement 35: In the Business Logic layer, a time comparison was conducted between the C# programming language and nMorph during the Analysis phase. The data suggest that participants generally took slightly more time when working with nMorph when compared to C#. On average, the time difference was noted to be 11.10 min. For instance, while Participant 1 spent 9 min on C#, they took 15 min with nMorph. Similarly, Participant 2 used 95.5 minutes for C# and 130 min for nMorph. However, it should be noted that Participants 3 and 5 exhibited no difference in time between the two languages.

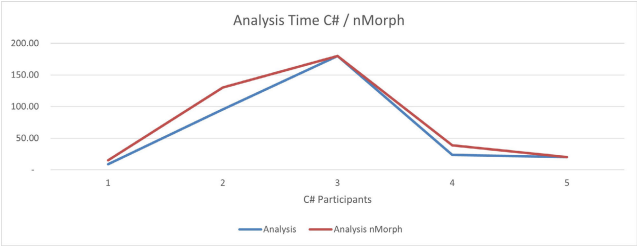


FIGURE 46. Measurement 35: Analysis time (C# - nMorph).

- Measurement 36: In the Business Logic layer, a time comparison was made between C# and nMorph during the Development phase. The data illustrate that, on average, participants spent an additional 4.95 minutes when working with nMorph compared to C#. For example, Participant 1 took 53.25 minutes with C# but only 22.5 minutes with nMorph. Conversely, Participant 4 spent 41.25 minutes on C# and 82.5 minutes on nMorph. However, it is important to highlight that both Participants 3 and 5 demonstrated no time difference in their use of the two languages.
- Measurement 37: In the Business Logic layer, a comparative analysis of the time spent during the Testing phase between C# and nMorph was conducted. From the gathered data, it emerges that participants, on average, took about 2.75 minutes longer with nMorph than they did with C#. Notably, Participant 1 completed testing more swiftly with nMorph, taking only 15 min,

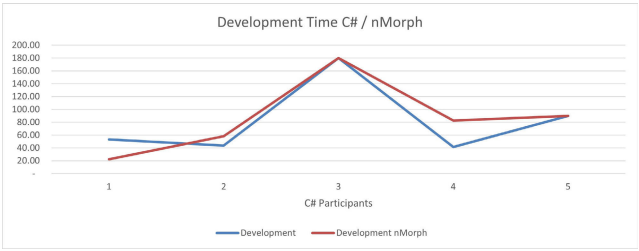


FIGURE 47. Measurement 36: Development time (C# - nMorph).

compared to 23.75 min with C#. On the other hand, Participant 3 took 60 min with C# and 75 min with nMorph. Meanwhile, Participants 2 and 4 also showed a slight increase in testing time when using nMorph. Participant 5, however, exhibited no difference in the time spent on the two platforms.

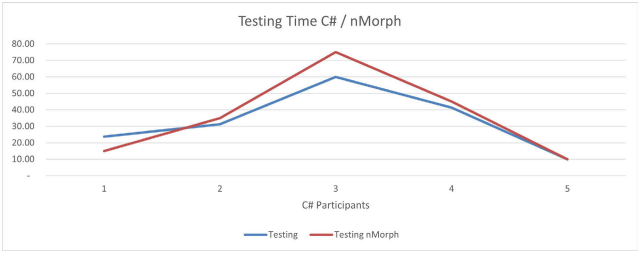


FIGURE 48. Measurement 37: Testing time (C# - nMorph).

- Measurement 38: Within the Business Logic layer, a comprehensive assessment was conducted comparing the total time taken by participants to complete the full cycle, encompassing Analysis, Development, and Testing phases, using C# and nMorph. The data suggests that, on average, participants took approximately 18.80 min longer when working with nMorph than with C#. For instance, Participant 1 finished the cycle more swiftly using nMorph, taking 52.5 min, in contrast to 86 min with C#. In contrast, Participant 2's time increased from 170.5 min with C# to 223 min with nMorph. Participants 3 and 4 also exhibited increased cycle times with nMorph, whereas Participant 5 maintained consistent times across both platforms.

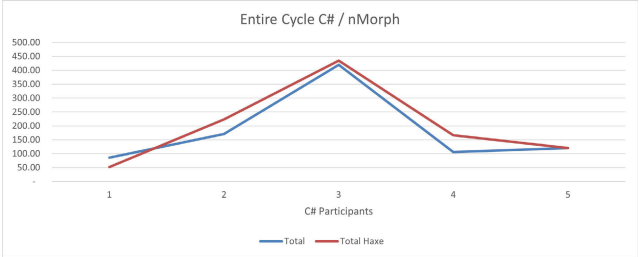


FIGURE 49. Measurement 38: Entire cycle time (C# - nMorph).

4) ABOUT JAVA ASSIGNMENT

Moreover, we could undertake detailed comparisons for participants who worked with Java, in contrast to nMorph. The main objective was to ascertain whether employing a distinct target technology, such as Java, and its associated learning process either eased or complicated the transition to nMorph.

- Measurement 39: Within the Business Logic layer, an evaluation was conducted focusing on the distribution of time across the different phases of construction using the Java programming language. The data indicate that the Development consumed the most significant portion of time at 61.88 min, which constitutes 59% of the total time. This is noticeably higher than that in other languages. The Analysis and Testing phases took 22.50 min (22%) and 20.00 min (19%) respectively. The prominence of the Development time suggests that Java might require a more extended developmental phase than its counterparts in this context.

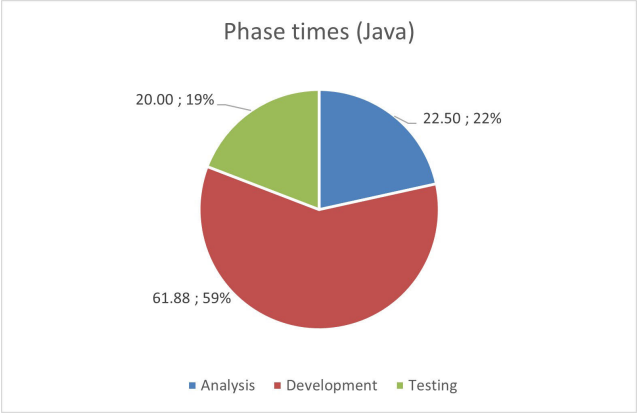


FIGURE 50. Measurement 39: Phase times (Java).

- Measurement 40: In the Business Logic layer, the time distribution for different construction phases was assessed for participants assigned to Java when using nMorph for the task. The data revealed that Development demanded the most substantial segment of time at 64.38 min, accounting for 60% of the entire duration. This is notably higher in comparison to other languages, echoing similar trends observed in Java. The Analysis phase took 26.56 min (24%), while Testing was completed in 16.88 minutes (16%). The relatively close proportions of time distribution across the phases suggest a balanced approach, but the dominance of Development time reinforces the idea that the nMorph might have inherent complexities in the development phase, at least for those originally familiar with Java.
- Measurement 41: In the Business Logic layer, a comparative analysis of the time taken during the Analysis phase was conducted between Java and nMorph. The data indicate that participants spent an additional 4.06 minutes when working with nMorph compared to Java. Specifically, Participant 1's analysis time increased by 2.5 minutes, Participant 2's by 2.5 minutes, Participant 3's, 2.5 minutes, and Participant 4, 8.75 minutes. This suggests that those familiar with Java might face a

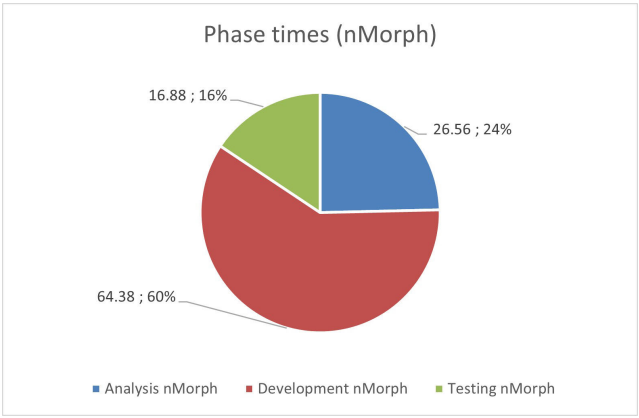


FIGURE 51. Measurement 40: Phase times (nMorph).

slightly steeper learning curve or require additional time to adapt to the nMorph environment during the analysis phase.

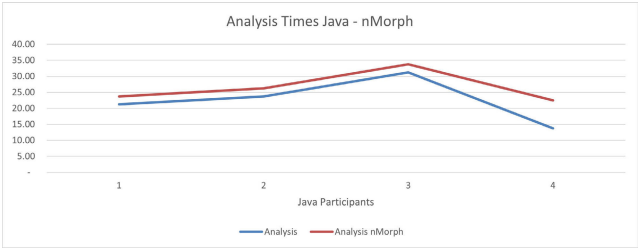


FIGURE 52. Measurement 41: Analysis times (Java - nMorph).

- Measurement 42: In the context of the Business Logic layer, a comparison was made between Java and nMorph regarding the time utilized during the Development phase. On average, participants spent an additional 2.50 minutes when working with nMorph compared to Java. Specifically, while Participant 1 and Participant 2 had relatively similar development times across both languages, Participant 3 was quicker by 7.5 minutes with nMorph, and Participant 4 took an additional 16.25 minutes when using nMorph. This suggests a slight variance in adaptability among the participants when transitioning from Java to the nMorph environment during the development process.

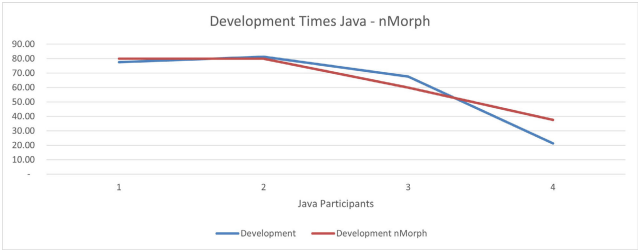


FIGURE 53. Measurement 42: Development times (Java - nMorph).

- Measurement 43: In the Business Logic layer, when examining the Testing phase, a comparison was made between the time taken by Java and nMorph. Interestingly, on average, participants consumed 3.13 minutes less when testing with nMorph than with Java. Breaking this down: Participants 1 and 2 maintained consistent times across both languages, while Participants 3 and 4 were faster with nMorph by 10 minutes and 2.5 minutes respectively. This indicates that in the testing phase, nMorph might offer efficiencies or familiarity that enable participants to complete tasks more swiftly than Java.

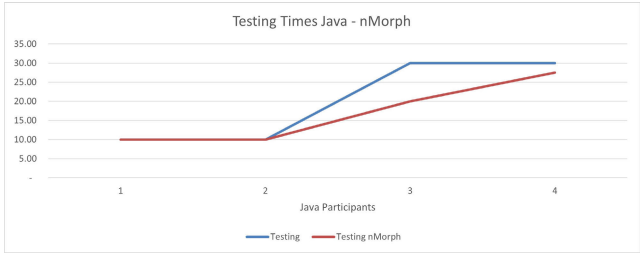


FIGURE 54. Measurement 43: Testing times (Java - nMorph).

- Measurement 44: In the Business Logic layer, during the complete cycle encompassing the Analysis, Development, and Testing phases, there was a comparison of time consumption between Java and nMorph. On average, participants took 3.44 minutes more when working with nMorph compared to Java. To detail this further: Participant 1 spent 5 minutes more on nMorph, Participant 2 used 1.25 minutes more with nMorph, Participant 3 actually used 15 minutes less on nMorph, and Participant 4 spent 22.5 minutes more with nMorph. This suggests that while nMorph may generally require slightly more time overall, individual experiences can vary, with some participants finding efficiency in certain aspects of the cycle with nMorph.

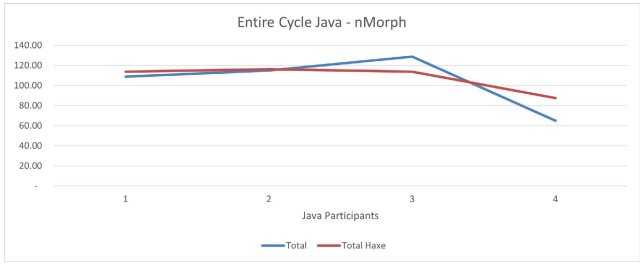


FIGURE 55. Measurement 44: Entire cycle times (Java - nMorph).

5) ABOUT PHP ASSIGNMENT

Additionally, it is valuable to conduct specific analyses for participants who were accustomed to PHP, and then compare their experiences with nMorph. This will help us understand whether familiarity with a particular technology, such as PHP, either simplifies or hinders their adaptation to nMorph.

- Measurement 45: In the Business Logic layer, when analyzing time allocation across the phases for PHP, it was observed that the participants devoted 36% of their time to Analysis, 41% to Development, and 23% to Testing. This distribution indicates that although

Development consumes the most time, there is a significant amount of time dedicated to Analysis, almost on par with Development. The least amount of time was allocated for Testing. This pattern provides insight into the demands of working with PHP in different stages of construction and suggests that both Analysis and Development require close attention and considerable time commitment.

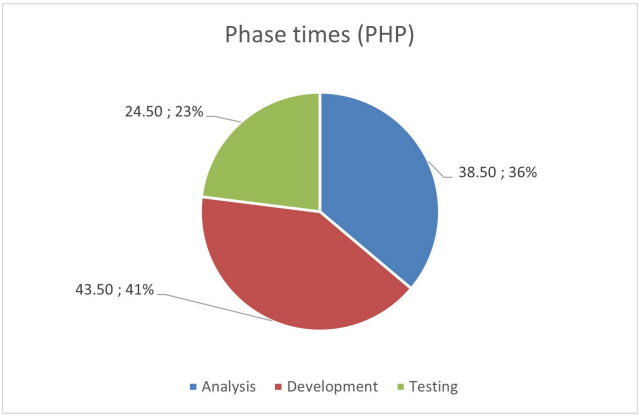


FIGURE 56. Measurement 45: Phase times (PHP).

- Measurement 46: Within the Business Logic layer, when observing the time distribution for the nMorph language among those who were initially assigned to PHP, the following allocations were noticed: 39% of the time was dedicated to Analysis, 42% to Development, and 19% to Testing. This distribution reveals a balanced time commitment between the Analysis and Development phases, with both demanding almost equal attention. However, the Testing phase required slightly less time to complete. The data suggest that when transitioning from PHP to nMorph, participants encountered a similar pattern of time allocation across phases, highlighting the comparative characteristics of the two languages in the construction process.

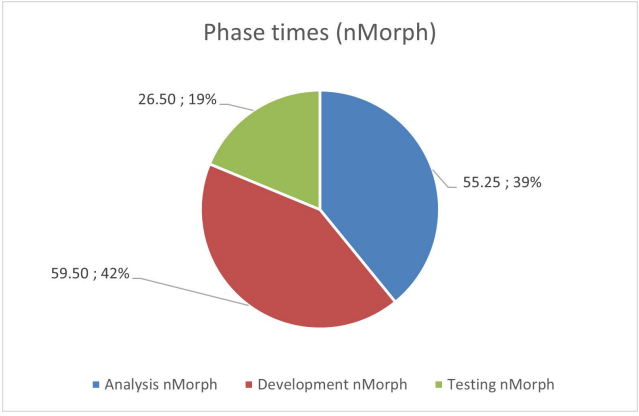


FIGURE 57. Measurement 46: Phase times (nMorph).

- Measurement 47: Within the Business Logic layer, a comparison of the time taken during the Analysis phase between PHP and nMorph was conducted. On average, participants took 16.75 minutes longer with nMorph than with PHP. More specifically, Participant 1 took an additional 11.25 minutes using nMorph, Participant 2 had a slight increase of 1.25 minutes, Participant 3 took an added 23.75 minutes, and notably, Participant 4 required an extra 47.5 minutes. However, Participant 5 remained consistent in both languages. These data underscore that while some participants adjusted seamlessly between languages, others faced more pronounced differences, contributing to the average increased time spent on nMorph during the Analysis phase.

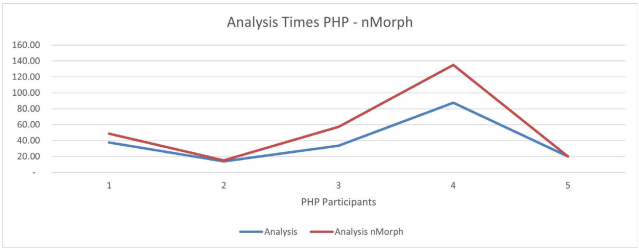


FIGURE 58. Measurement 47: Analysis times (PHP - nMorph).

- Measurement 48: In the Business Logic layer, a comparison of the time taken during the Development phase between PHP and nMorph reveals some distinct variations among participants. The average additional time spent with nMorph compared with PHP was 16.00 min. To break it down: Participant 1 needed 7.5 min more with nMorph, while Participant 2 spent 18.75 min less. Participant 3 took a notable 37.5 min longer with nMorph, Participant 4 required an additional 45 min, and Participant 5 used 8.75 min more. These fluctuations indicate that participants' adaptability to nMorph varied considerably, with some finding it more time-consuming and others actually experiencing a decrease in development time when switching from PHP.

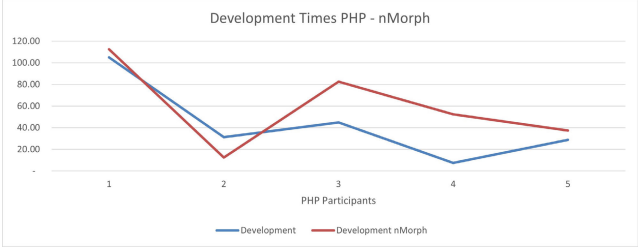


FIGURE 59. Measurement 48: Development times (PHP - nMorph).

- Measurement 49: In the Business Logic layer, a comparison was made between the time consumed during the Testing phase for both PHP and nMorph. On average, participants took 2.00 minutes longer when using nMorph than they did with PHP. Diving deeper: Participant 1 took 3.75 minutes longer with nMorph, while Participant 2 finished 5 minutes faster. Participant 3 needed an additional 12.5 minutes with nMorph,

Participant 4 took 1.25 minutes more, and Participant 5 used 2.5 minutes less. This suggests that while some participants found testing in nMorph to be slightly more time-intensive, others actually experienced a reduction in the time required compared to PHP.

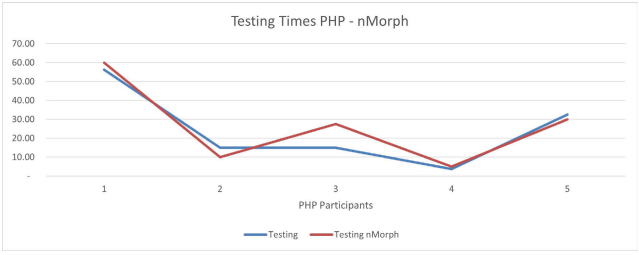


FIGURE 60. Measurement 49: Testing times (PHP - nMorph).

- Measurement 50: In the Business Logic layer, the overall development cycle time was compared between PHP and nMorph. On average, using nMorph resulted in an additional 34.75 minutes compared to PHP. From the provided data: Participants 1, 3, 4, and 5 recorded more time with nMorph; Notably, Participant 2 took less time using nMorph; This observation suggests that nMorph often requires more time, but the individual outcomes may differ.

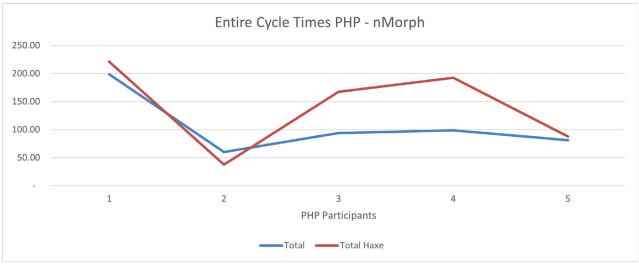


FIGURE 61. Measurement 50: Entire cycle times (PHP - nMorph).

6) GLOBAL RESULTS

Measurement 51: The data indicate variations in time across different phases when comparing the nMorph language with others. Moreover, the data included variables related to the perceived difficulty and understanding of tasks.

Key Insights: - In the analysis phase, the times for C# and PHP were comparable, taking roughly a third more time than their baseline when using nMorph. Java exhibited a 23.48% increase. - The development phase showed a stark contrast: while C# and Java had modest time increases with nMorph, PHP's development time skyrocketed by over 132%. - In testing, Java displayed a 10.42% decrease in time with nMorph, whereas C# had a minor increase and PHP had a 16.46% boost. - For the entire development cycle, PHP participants experienced the greatest time increase at 31.02%, with C# and Java trailing at 10.38% and 7.16% respectively. - Regarding perceived difficulty, nMorph appeared to be the most challenging for the C# participants, with a 49.75% increase. PHP trailed behind at 35.85%, and Java was deemed significantly less difficult at 11.25%. - In terms of understanding, C# and Java participants found nMorph somewhat more challenging, as reflected by negative percentages. Meanwhile, PHP participants demonstrated a slight increase in understanding with a 10.03% boost.

The data suggest that while transitioning to nMorph, those accustomed to PHP experienced the most significant adjustments in development time, even though their understanding improved. On the other hand, C# users experienced the steepest increase in difficulty, despite the relatively modest time increase.

	C#	Java	PHP
Análisis	33.19%	23.48%	32.75%
Desarrollo	14.96%	16.76%	132.18%
Pruebas	1.85%	-10.42%	16.46%
Tiempo total de ciclo	10.38%	7.16%	31.02%
Dificultad Percibida	49.75%	11.25%	35.85%
Entendimiento	-9.30%	-4.37%	10.03%

FIGURE 62. Measurement 51: Time variations.

7) OPEN-ENDED QUESTIONS

Upon analyzing the responses to the open-ended questions, the following main criteria were identified:

- Most participants did not report any issues or remarkable events during the exercise execution.
- There was a notable to mention about the lack of information available online to address infrequent problems. Participants commented due to the lack of documentation. Although this criterion was explicitly mentioned, it was only a concern for two participants. Supplementary documentation was not required for the exercise execution. This could suggest that there might have been oversight in reviewing the materials provided before the exercise by these specific participants.
- Several participants mentioned having faced minor issues but were able to find solutions relatively easily.
- The feedback emphasized that the provided materials and corresponding training were highly beneficial in achieving the goals of the proposed exercise.

G. EXPERIMENT INSIGHTS

1) OUTCOMES

This study utilized a quasi-experimental approach to assess the efficacy of the nMorph framework for novice software developers. Data were gathered from 14 advanced students from a software programming course. Participants were randomly assigned to a programming language: C#, Java, or PHP (AL). A preliminary educational process was implemented to equalize their understanding of these languages and familiarize them with Haxe, which is essential for the nMorph framework. Following this, participants executed a programming task twice, once using the assigned language and other with Haxe and nMorph framework. During this exercise, they monitored themselves their timings and perceptions. 51 specific measurements were collected and analyzed in-depth.

Experiment Research Questions Analysis:

- *To what extent is the nMorph framework suitable for novice developers' business logic coding?*
  The programming task was executed successfully, demonstrating the full applicability of the nMorph framework for business logic coding within the context of a disposable prototype.
- *Which comparative parameters are key to determining the efficiency and effectiveness of the nMorph framework compared with the direct application of a target language?*
  Key findings related to efficiency and effectiveness based on participants' perceptions during exercise are as follows.
  - Finding 1: There was no significant correlation between Perceived Difficulty and timing for analysis, development, testing, or the entire cycle. This suggests that perceived complexity does not necessarily dictate the duration of task execution, regardless of whether the assigned languages or Haxe with nMorph are used.
  - Finding 2: No significant correlation was observed between Language Proficiency and timing for analysis, development, testing, or the entire cycle. This implies that higher proficiency in the assigned languages or Haxe does not necessarily lead to faster or slower task execution.
  - Finding 3: There was no significant correlation between Task Understanding and timing for analysis, development, testing, or the entire cycle.
  - Finding 4: Several perception variables were found to be significantly correlated. Higher Language Proficiency is associated with lower Perceived Difficulty. Higher Language Proficiency leads to a better Task Understanding. Greater Task Understanding was associated with a lower Perceived Difficulty. This correlation was more pronounced for the assigned languages than for Haxe.
  - Finding 5: On average, participants spent 33% of their time in the analysis phase, 46% in the development phase, and 21% in testing phase. A specific variation was observed for participants using Java, who spent a greater proportion of their time in the development phase.
  - Finding 6: Haxe development took longer than any of the assigned languages, with an estimated average variation of 15%.
  - Finding 7: There was a pair-wise correlation between Analysis time with the assigned language and nMorph, Development time with the assigned language and nMorph, and Testing time with the assigned language and nMorph.

  In summary, utilizing nMorph with Haxe would, require an additional 15% execution time compared to other commonly used programming languages in enterprise applications (C#, Java, and PHP). No other variables directly influencing the measured efficiency and effectiveness metrics were identified.
- *What programming practices are applied by programmers using the nMorph framework in comparison with those used in other languages?*
  The objective of the exercise was to evaluate the business logic coding executed by participants. It was observed that for the C# and Java languages, the solutions were more structured, defined as projects, and utilized object-oriented programming. For PHP, the solutions appeared simpler, generally consisting of single files with linear programming.
  Regarding database connection, participants assigned to C#, predominantly used the Entity Framework. Those assigned to Java used JPA and direct connections. PHP participants mostly employed direct connections, with one individual using Propel. Not all participants employed an ORM for database connectivity; several opted for direct query operations.
  For programming with Haxe, the source code structure was consistent across all participants. All utilized the basic programming template provided by the nMorph framework and engaged with the ORM provided by nMorph. A general deficiency was noticed in the inclusion of explanatory comments within the implemented logic.
  The developed algorithms showcased various approaches to the proposed problem. Except for one participant, all participants achieved the goal of data manipulation in the assigned database tables and accomplished all transactions. The majority employed reusability in implementing the money transfer transaction between accounts, which involved both a withdrawal and a deposit. This was observed in both the assigned language and with Haxe.
  Not all participants provided adequate validation in their developed transactions. The applied validations were overly simplistic and did not address several potential issues, with a few exceptions.
  Regarding the user interface, participants were instructed to develop a textual console solution as the focus of the study was on their business logic development capability. Nevertheless, some participants opted for native technologies in the assigned language, such as asp.net for C# or Spring for Java. These decisions may introduce variability in the execution time of the exercise.
  While each participant applied technology based on their interpretation, it was confirmed that the programming logic in both the assigned language and Haxe met the expected algorithm and functional objectives for the exercise. Functional implementations equivalent to those achieved with the assigned programming languages were attainable with the nMorph framework and Haxe.

- *What challenges or difficulties were encountered during the study that could enhance the framework for future exercises?*
  The primary challenge was unfamiliarity with the technologies used in the exercise. Although participants received comprehensive training in the assigned language and detailed lectures on Haxe and the nMorph framework at onset of the exercise, proficiency in these technologies requires time and experience. The execution time of the exercise did not allow extended educational efforts. For future exercises, earlier training on related technologies, especially regarding the nMorph framework, will help to reduce knowledge gaps. Technically, a difficulty noted was the lack of specific documentation for the technologies utilized. While abundant documentation is available for the assigned programming languages, Haxe's is more limited, and nMorph's is virtually non-existent as its operational foundation and dissemination are still under research development. Although the education sessions and videos provided aimed to mitigate this factor, some participants remained concerned. It is crucial for the framework to have comprehensive, organized documentation to ensure developers have adequate support and confidence in developing their business logic.
  The measurement strategy relied on a data form that captured the developer's perceptions of the exercise. This method introduces subjectivity, possibly skewing data. For a more comprehensive efficiency and efficacy assessment, metrics should not solely rely on participant self-recording, but also employ external supervisory mechanisms and data collection tools.
  Despite the proposed exercise was very useful to understand the adoption of a first group of developers, it is important to note that it was limited and straightforward. The collected data may differ considerably in larger, longer projects. The information serves as a reference for those new to these technologies but should not be considered definitive concerning the nMorph framework's application. Various complementary methods should be sought to establish better applicability and validity by incorporating real-world scenarios, case studies, dissemination elements, and others.

2) REFLECTIONS ON FINDINGS

The investigation into ''Business Logic'' provides insightful data regarding the relationship between analysis time and perceived task difficulty, suggesting that tasks deemed more challenging take longer to analyze. Intriguingly, language familiarity exhibited a negative correlation with analysis time, indicating that those with greater language fluency can analyze tasks more swiftly. However, language familiarity did not appear to significantly impact development time. When examining the testing phase, a correlation was noted among language knowledge, task comprehension, and test time in the assigned language. For the Haxe-nMorph transpiler, however, only task comprehension showed a correlation with testing time.

Considering language disparities, participants with higher knowledge of the assigned language found the tasks less daunting, resulting in better task understanding. Participants generally took longer in the analysis phase when employing Haxe as opposed to the assigned language, averaging an additional 20.11 minutes across all tasks with Haxe.

Findings from the ''User Interface'' domain indicate a marginal correlation between programming language familiarity and analysis time during the UI phase. When using the Haxe-nMorph transpiler, a moderate increase in analysis time was observed as the perceived difficulty increased. Despite these variations, the distribution of time across the analysis, development, and testing phases remained fairly consistent, irrespective of the language employed.

Regarding specific language assignments, such as C# and Java, insights revealed variations in time allocation across phases. For instance, those assigned to C# followed by nMorph exhibited an equitable distribution of effort across phases. However, when comparing nMorph with C# and Java, differences in the time spent in each phase were discerned. However, PHP analysis revealed that although PHP and nMorph shared time distribution parallels, transitioning from PHP to nMorph often resulted in added time requirements, especially during the analysis and development phases.

In global outcome terms, the data suggest that those accustomed to PHP experienced the most significant shifts in development time when switching to nMorph. While understanding improved, C# users experienced the steepest increase in task difficulty, despite a modest time increment.

These insights offer a clear understanding of how language familiarity and comprehension can influence various development process phases and underscore efficiency disparities between the assigned language and the Haxe-nMorph transpiler.

IV. DISCUSSION

A. ABOUT THE RESEARCH QUESTIONS

- *RQ1: In the context of multi-programming language software development, how does the nMorph framework's transpiler-based approach demonstrate feasibility and efficacy compared to traditional programming languages?*
  In the evolving realm of multi-programming language software development, the nMorph framework has emerged as a beacon of innovation, offering a comprehensive solution to the contemporary challenges of adaptability, scalability, and performance. Rooted in a transpiler-based architecture, the nMorph framework introduces a transformative approach that empowers developers to code in a singular language. This code can then be transpiled flawlessly into a multitude of target languages, ensuring the preservation of the integrity of the core design.

The true demonstration of its feasibility lies in nMorph's meticulous design, which is tailored to serve enterprise back-ends. Through transpilable languages, it provides assurance of unwavering performance, irrespective of platform-specific compilations. Its architecture, laden with features such as ORM & Database Connectors, and a security layer, is indicative of its efficacy in handling various back-end responsibilities with finesse. Furthermore, the back-end functionalities in the framework can be expressed through diverse methods, all supported by state-of-the-art tools, such as the PostBuild Utility and DB Gen component.

The selection of the Haxe transpiler as the nucleus of the framework emphasizes its compatibility prowess, especially with dominant enterprise languages such as Java, C#, and PHP. This choice alone showcases the framework's pragmatic direction, ensuring it aligns with the current technological landscape, supported by databases like Oracle Database, SQL Server, and MariaDB, as well as servers such as IIS and Apache.

The nMorph framework's transpiler-based approach offers a unique perspective in its comparison with traditional programming languages. An empirical experiment conducted on throwaway prototypes offers a clear perspective on its real-world viability. Novice developers, when exposed to both traditional languages and the Haxe language with nMorph, provided insights that underscored the framework's potential while highlighting certain challenges. Despite the observed variations in perceived difficulty and the slightly extended time participants took with Haxe, the application of nMorph framework in business logic development remains promising. Notably, these challenges can be mitigated through enhanced training and documentation.

The experiment results revealed that while transitioning to nMorph, the time allocated across different development phases varied depending on the original programming language in use. For instance, developers accustomed to PHP experienced the most significant adjustments in development time, with a marked 31.02% increase for the entire cycle, despite an improvement in their understanding. Conversely, C# participants, although witnessing a comparatively modest time increase of 10.38%, reported a substantial rise in perceived difficulty by 49.75%. Java developers showed nuanced transitions, with an increase in analysis and development times, but a time-saving of 10.42% in the testing phase when using nMorph.

Furthermore, when analyzing the Business Logic layer, it was observed that the nMorph framework led to longer analysis and development times in some cases, with developers consistently dedicating significant portions of their time to the development phase regardless of the programming language. Yet, there were inherent similarities in time allocation and correlations between traditional languages and nMorph during various development phases. This suggests that while nMorph introduces certain challenges, especially in terms of perceived difficulty and time allocation in some phases, it still aligns well with conventional programming paradigms. The insights underscore nMorph's feasibility in integrating with existing workflows while highlighting areas that might demand more attention or optimization in its transpiler-based approach compared to traditional programming languages.

The quasi-experimental design of this study, which encompassed novice developers, indicates that while the nMorph framework is entirely applicable for business logic coding within prototyping contexts, its integration with Haxe does present an additional execution time of approximately 15% in comparison to traditionally used enterprise languages such as C#, Java, and PHP. Key metrics such as Perceived Difficulty, Language Proficiency, and Task Understanding, intriguingly, did not exhibit significant correlations with the timing of various development phases, emphasizing that time expenditure may not necessarily align with perceived complexities or proficiencies. Furthermore, it's noteworthy that even though participants assigned to different languages like C# and Java displayed structured, project-defined solutions with object-oriented programming, their transition to Haxe using the nMorph framework yielded solutions with consistent source code structures. However, proficiency in these technologies, especially nMorph, remains a challenge due to the preliminary stage of its documentation and its relative novelty in the development community. Technical barriers notwithstanding, the functionality and outcomes achieved with the nMorph framework and Haxe mirrored those achieved with traditional programming languages. Thus, while the nMorph transpiler-based approach underscores potential and viability in multi-language software development, its efficacy, in terms of time efficiency, remains moderately lower than that of its traditional counterparts.

In conclusion, the nMorph framework, with its transpiler-based approach, not only demonstrates feasibility in the multi-programming language software development sphere but also shows a marked efficacy when juxtaposed with traditional programming languages. Although there are areas for improvement, the advantages it offers in terms of customization, technological longevity, and brand independence make it a compelling choice for the future of software development.

- *RQ2: How does the implementation of the nMorph framework influence the experience and efficiency of developers, especially in comparison to commonly used programming languages (C#, PHP, or Java) for business logic development?*

  In the realm of multi-programming language software development, the nMorph framework introduces a

distinctive transpiler-based architecture, enabling developers to code in one language and then transpile it into multiple target languages, ensuring the preservation of the core design. A few key observations emerge when analyzing its influence on novice developers.

Firstly, the design of nMorph framework focuses on catering to enterprise back-ends with its layered architecture that encompasses various functionalities, from ORM & Database Connectors to data access layers. This sophisticated design, is meant to guarantee consistent performance across different platform-specific compilations. For novice developers, navigating such an expansive framework may present an initial learning curve, particularly if they are familiar with traditional programming languages.

This empirical study provides valuable insights into the framework's influence on novice developers. When tasked with implementing business logic using both the Haxe language with nMorph and their designated programming languages (C#, PHP, or Java), there were evident variations in their experiences. Participants took, on average, 20.11 minutes longer with Haxe and the nMorph framework than with their designated language. Specific challenges arose depending on the traditional language they were using, for instance, PHP users faced significant time shifts in development, whereas C# users found the framework particularly challenging.

These data accentuate the importance of individual developer proficiency and familiarity with the language in influencing software development outcomes. Although nMorph offers a revolutionary approach, its implementation seems to extend the development phases for novice developers compared to traditional languages. A pivotal aspect of this extension is the perceived task difficulty, which has a direct correlation with extended analysis durations. Interestingly, while language proficiency influenced the speed of analysis, it seemed to have a limited impact on overall development time.

In terms of efficiency, the introduction of the nMorph framework seems to generally increase the development time across the board when compared with the commonly used languages, specifically C#, PHP, and Java. For instance, developers transitioning from PHP to nMorph experienced the most significant increase in development time across the entire cycle, averaging an additional 34.75 minutes. When comparing the entire development cycle, PHP participants exhibited the most substantial time increase at 31.02%, with C# and Java trailing at 10.38% and 7.16%, respectively. On the other hand, some phases, like the testing phase for Java developers, showed improved times with nMorph, suggesting specific efficiencies within certain aspects of the development process.

Regarding the experience, the perceived difficulty of nMorph varied among participants with different primary languages. C# participants found the transition to nMorph to be the most challenging, reflecting a 49.75% increase in perceived difficulty. In contrast, Java participants deemed the framework to be significantly less troublesome with only an 11.25% increase in difficulty. Notably, despite the longer time demands with nMorph, PHP participants showed an enhanced understanding, suggesting that while the framework might be more time-intensive, it could potentially offer clearer or more comprehensible constructs, at least for those transitioning from PHP.

Moreover, certain perceptions were tied to the programming language and the framework. Higher language proficiency was linked to lower perceived difficulty and enhanced task understanding. However, this understanding didn't necessarily translate to expedited task execution. For instance, a deep understanding of C# or Java did not always lead to faster development times. Additionally, participants using Java spent a notably larger proportion of their time in the development phase, while those accustomed to PHP observed the most considerable shifts in development time when transitioning to nMorph. The study further highlighted a consistent approach among developers using the nMorph framework, which was evident in their adherence to the basic programming template and the utilization of the ORM provided by nMorph. While solutions crafted in C# and Java were more structured and typically followed object-oriented programming principles, those in PHP were simpler and often linear. A commonality across all languages, however, was the emphasis on business logic coding, with most participants achieving the intended data manipulation in the assigned database tables. In essence, while the nMorph framework facilitates achieving equivalent functional implementations, it necessitates a longer execution time compared to traditional languages, underscoring a nuanced trade-off between familiarity, efficiency, and adaptability in the realm of business logic development.

In conclusion, the nMorph framework, while promising, influence the experience of novice developers by presenting a steeper learning curve and extending development times when compared to traditional programming languages for business logic development. However, these challenges also point towards the necessity for more robust training and documentation to aid novice developers in transitioning smoothly between languages and frameworks.

- *RQ3: Considering the transition between traditional programming approaches and the nMorph framework, to what extent does a developer's language proficiency impact adaptability, scalability, and performance outcomes in software construction and practice?*

  A empirical experiment was conducted with a group of developers proposing several measurements. Several insights arise:

1) *Adaptability*: The experiment showed a varied experience for developers based on their language proficiency. Participants took an average of 20.11 minutes longer when using Haxe-nMorph then when using their familiar designated language. Such findings underscore the potential challenges developers might face in adapting to the nMorph framework, especially if their proficiency in the designated language is high. However, it is worth noting that adaptability is not solely dependent on language proficiency; task comprehension and perceived difficulty also play a crucial role.
2) Scalability: While the study's primary focus was not on scalability outcomes, the multifaceted architecture of the nMorph framework, with its diverse responsibilities and tools, indicates a design aimed at scalable solutions. However, if developers are not proficient in understanding and implementing the framework, this can impact the scalability of the constructed software because of potential errors or inefficient code practices.
3) Performance: Language proficiency was shown to influence the speed of analysis but had a minimal direct impact on the overall development time. This suggests that while a developer's language proficiency can influence the initial phases of software construction, it might not have a substantial direct impact on the ultimate performance of the software. However, indirect impacts, such as potential errors or less optimized solutions due to unfamiliarity, could affect performance outcomes.

A developer's language proficiency influences adaptability, scalability, and performance outcomes and it is crucial when transitioning between traditional programming and the nMorph framework.

It's evident that a developer's familiarity with a particular language plays a pivotal role in adaptability when transitioning to the nMorph framework. When observing time allocation, participants from the PHP background experienced the most significant shifts in development time, with an overall increase of 31.02% when using nMorph. Java developers, on the other hand, displayed varied outcomes, with an average increase of 7.16% across all phases, yet outperforming during the testing phase. C# developers experienced a 10.38% increase in time across the cycle, highlighting a moderate adjustment required. However, perceived difficulty was most pronounced for C# participants, registering a 49.75% increase, suggesting that while the time difference might not be as drastic, the complexity and challenge faced were more evident. Conversely, PHP developers, despite the time challenges, showcased a 10.03% improvement in understanding, implying that their adaptability was gradually improving.

Moreover, while the scalability and performance outcomes varied across languages, inherent trends emerged. In terms of development phase, the shift to nMorph from traditional languages like PHP showed a stark contrast with an over 132% increase in time, suggesting potential scalability concerns. However, Java developers experienced a decrease in testing time by 10.42%, hinting at potential performance efficiencies when scaling with nMorph. The overall adaptability, as reflected by understanding and perceived difficulty metrics, indicates that each language background brings its unique challenges and advantages when transitioning to nMorph. Thus, a developer's language proficiency profoundly impacts the outcomes in terms of adaptability, scalability, and performance when integrating with the nMorph framework.

In examining the transition from traditional programming paradigms to the nMorph framework, language proficiency emerges as a pivotal factor influencing adaptability, scalability, and performance outcomes in software construction and practice. The findings illustrate that participants' familiarity with the assigned language, whether it be C#, Java, or PHP, does not necessarily correlate with the duration of task execution, suggesting that proficiency in the traditional programming languages does not guarantee quicker adaptability to the nMorph framework. Furthermore, while higher language proficiency was associated with reduced perceived task difficulty, this did not directly translate into efficiency gains in the use of Haxe with the nMorph framework. Remarkably, a more profound understanding of the assigned language led to better task comprehension, but this advantage diminished when transitioning to Haxe, indicating a potential adaptability challenge. With respect to scalability, participants using traditional languages like C# and Java exhibited structured, project-based solutions that leveraged object-oriented programming. In contrast, Haxe solutions, under the nMorph framework, showed a consistent source code structure across all participants, suggesting a standardized approach that might offer scalability benefits in larger team settings. Performance-wise, while developers proficient in C# or Java experienced variations in time allocation across different development phases, those accustomed to PHP witnessed the most pronounced shifts when transitioning to nMorph, especially during the analysis and development phases. This variance underscores the differential impact of language proficiency on performance outcomes when transitioning from traditional programming approaches to the nMorph framework. In essence, while language proficiency influences task comprehension and perceived difficulty, its direct impact on adaptability to the nMorph framework varies, and its implications for scalability and performance are multifaceted.

In conclusion, while the nMorph framework presents a promising approach to multi-platform software development, a developer's language proficiency plays a significant role in adaptability, influencing the initial phase of software construction. Its effect on scalability and performance, though indirect, can arise from potential pitfalls or inefficiencies due to a lack of familiarity. Thus, for optimal outcomes in adaptability, scalability, and performance, a balance between language proficiency and comprehensive understanding of the framework is essential. This study also highlights the importance of intensive training and robust documentation to support developers in making this transition.

B. ABOUT PREVIOUS ARTICLES BY THE AUTHORS

Given the vast scope of the nMorph framework's introduction and its empirical experiment, understanding the authors' previous researches provides an invaluable context. Let us delve into the authors' prior work and their relationship with the nMorph framework.

- *Transpiler-Based Architecture for Multi-platform Web Applications [18]*
  Relevance to the Current Study: This initial research was the foundation for the nMorph framework, centering on a transpiler-based architectural design for building multi-platform web applications. It directly correlates with nMorph's principle of writing code in one language and transpiling it into multiple languages, with a focus on C#, Java, and PHP. Using the Haxe transpiler, the experimental prototype can be seen as a precursor to the nMorph framework.
  Insight & Contribution: By presenting preliminary results of using Haxe for transpiler-based architecture, this article solidifies the basis upon which the nMorph framework was built.
- *A systematic review on Transpiler usage for Transaction-Oriented Applications: [19]*
  Relevance to the Current Study: This systematic literature review highlights the gaps in the existing literature regarding transpiler implementations for transaction-oriented applications. The nMorph framework, which is essentially a transpiler-based methodology, is a direct response to the needs identified in this review.
  Insight & Contribution: By determining that transpilers had not been previously used for multi-platform transaction-oriented applications, the authors identified a significant opportunity. This article not only provided the academic rationale for the nMorph framework's inception, but also established the significance of multi-platform and multi-programming-language software architecture for better adaptability in software development.
- *Transpilers: A Systematic Mapping Review of Their Usage in Research and Industry [8]*
  Relevance to the Current Study: The systematic mapping review provides a comprehensive overview of how transpilers have been employed in diverse sectors over a decade. The vast applications, from AI, and graphics, to software development, underscore the versatility of transpilers. The intention of the nMorph framework to transform multi-platform software development fits within this larger narrative, particularly as the article also highlights transactional software and multi-platform applications as future research areas for transpilers.
  Insight & Contribution: This article paints a broader picture of transpilers in both academia and industry, hinting at their rising importance. The suggested areas for future research not only set the stage for specialized tools such as the nMorph framework, but also emphasize the framework's timely introduction in the context of current trends and demands.
- *Transpiler-Based Architecture Design Model for Back-End Layers in Software Development [5]*
  Relevance to the Current Study: This study serves as a keystone for the nMorph framework. While the aforementioned research touched upon a transpiler-based architectural design for multi-platform web applications [18], this study dives deeper into the integration of transpilers into the back-end layer. It proposes a complete methodology, from abstract concepts to detailed designs, for the automated transformation of business logic from one source code into multiple programming languages. The nMorph framework is an evolution or enhanced manifestation of this model, focusing on multi-platform back-end software development.
  Insight & Contribution: By introducing both the conceptual and practical elements of using transpilers in software design and architecture, this study sets the groundwork for the nMorph framework. The collaborative To-do-list application provides a hands-on example of the model's applicability, and more importantly, its feasibility. Thus, while this paper discusses the architecture model, the nMorph framework appears to be its tangible realization, incorporating the lessons learned from this initial implementation.

The tapestry of the authors' research trajectory, starting from the significance of transpilers, moving to their specific architectural model and culminating in the nMorph framework, offers an insightful journey. Each work builds upon the previous, with the "Transpiler-Based Architecture Design Model for Back-End Layers in Software Development" [5] serving as the direct precursor to the nMorph framework. While initial research established the landscape and gaps in the field, this particular article sets the theoretical and practical framework, upon which the nMorph was crafted, to address the challenges of modern software development. The combined body of work represents a meticulous progression, with each step paving the way for the next, in the quest for enhanced adaptability, scalability, and performance in software development.

C. ABOUT RELATED ARTICLES

The article on ''Compiler-compiler of multi syntax programming languages '' [20] emphasizes ensuring software reliability through N-version programming, where redundancy arises from software components' diversity. In contrast, this study aims to enable developers to code in a unified language, subsequently transpiling it into multiple target languages, thus harmonizing modern software development needs with the complexities of growing digital systems. The former article proposes a multi-syntax programming language compiler rooted in existing flex and bison specifications to facilitate error detection. This approach leans on the Haxe transpiler, tailoring the framework for enterprise back-end development, and integrating features like ORM and Database Connectors. An empirical study on the nMorph framework indicated participants took longer using it compared to traditional languages, but benefits of multi-language outputs counterbalanced this. The ''Compiler-compiler'' article focuses on simplifying error detection in multi-syntax software for industrial applications. In comparison, the nMorph framework suggests future exploration of its real-world applications, with an emphasis on refining based on user feedback and in-depth case studies.

The ''Multi-programming language software systems modularization'' article [21] recognizes the challenges posed by the deviation of evolving software systems from their original structures. A key emphasis is on understanding large-scale applications written in multiple languages, such as Mozilla Firefox 3.7, which employs over twenty programming languages. In comparison, the current study focuses on providing developers with a unified platform to code and then transpile into various target languages, retaining the software's core integrity and addressing the intricacies of modern digital systems. The modularization article introduces the concept of multi-programming language modularization, proposing a technique to comprehend applications composed in multiple programming languages. This is aimed at aiding the understanding of complex applications. The nMorph framework, on the other hand, leans on the Haxe transpiler, tailored especially for enterprise back-end development. It integrates features like ORM, Database Connectors, and a dedicated security layer, facilitating smooth transitions between languages such as Java, C#, and PHP while ensuring compatibility with multiple databases and servers. The modularization article showcases the applicability of their approach on Mozilla Firefox, highlighting its capability to yield a modularization close to what human experts would produce. The nMorph study conducted a real-world usability experiment, which indicated certain challenges in transitioning to nMorph, primarily contingent upon a developer's foundational language. Despite the longer time taken, the multi-language and multi-platform benefits counterbalance the investment. The nMorph research underscores the importance of continuous refinement through feedback and detailed case studies.

The ''Comparison of the most popular object-oriented software languages and criterions for introductory programming courses with analytic network process: A pilot study'' article [22] addresses the challenge of selecting the appropriate programming language for introductory courses. This decision-making process is multi-faceted, involving various closely interrelated factors. The study leverages the Analytic Network Process (ANP) methodology, aiming to simplify and streamline the decision-making process in the context of these multiple criteria. In contrast, this article introduces a paradigm shift. With the nMorph framework, the traditional challenge of selecting a single programming language becomes obsolete. Instead of being confined to one language choice, developers are empowered to code in a unified language, which can then be transpiled into multiple target languages. This maintains the software's original integrity while providing flexibility across platforms. The first article uses ANP to evaluate and rank programming languages based on a set of predefined criteria. The approach is analytical, focusing on a weighted decision-making process that accounts for the interactions between the criteria. The nMorph framework, however, centralizes around the Haxe transpiler. This innovative toolset allows for seamless transitions between popular languages like Java, C#, and PHP. Furthermore, the framework integrates essential features tailored for enterprise back-end development, ensuring compatibility with various databases and servers. The ANP-based study provides stakeholders with a structured method to assess and rank programming languages, potentially guiding educational institutions in crafting their introductory programming courses. However, it operates under the traditional assumption that one must choose a singular language. On the other hand, the nMorph framework liberates developers from this constraining decision. The multi-language, multi-platform outputs that nMorph facilitates render the initial language selection unnecessary. Developers can implement software solutions in parallel across platforms without the traditional barriers. Thus, with nMorph, the emphasis shifts from ''which language to choose'' to ''how to effectively leverage multi-language capabilities.''

D. THREATS TO VALIDITY

Understanding the validity of the research is paramount in gauging the reliability and generalizability of its findings. In evaluating our study on the nMorph framework, we identify potential threats across internal, external, and construct validity and also elucidate the measures taken to mitigate these threats.

1) Internal Validity
   - Sample Size and Characteristics: The limited sample of 14 students from the same software development university program may not be broad enough to draw conclusive insights. Their novice status further exacerbates this concern. Mitigation: Despite the small sample size, the participants were carefully chosen to represent a diverse

range of proficiency levels within their academic cohort. This was done to capture a spectrum of experiences and reduce bias stemming from proficiency variance among students.

- Laboratory Scenario: The controlled environment of the laboratory might not reflect the real-world intricacies and challenges faced in practical software development. Mitigation: Although the study was executed in a controlled environment, the assigned tasks were modeled after real-world challenges. The banking transaction exercise was designed to emulate a common development project, with the aim of making the experience as authentic as possible.

2) External Validity:

- Generalizability: Given the nascent status of the nMorph framework as a non-generally-available product, the broad applicability of the results is potentially constrained. Mitigation: We emphasize the exploratory nature of this study and advocate caution when attempting to generalize its outcomes. The primary goal was to gain initial insights rather than provide a universal conclusion.
- Participant Demographics: This study's focus on novice developers from a single university could hinder the relevance of the findings to professionals and other demographic groups. Mitigation: The research was explicit in its objectives, which were centered on gauging the adaptability of novice developers to the nMorph framework. Future research is planned to involve a more diverse participant pool, including professional developers, to enhance external validity.

3) Construct Validity:

- Measurement Tools: Relying predominantly on post-implementation surveys and source code analysis might lead to an incomplete understanding of the participants' experiences. Mitigation: By integrating both qualitative (surveys) and quantitative (source code analysis) tools, we attempted to capture participants' experiences holistically. Additionally, the researchers maintained open communication channels with participants, encouraging them to share any challenges or insights beyond the structured survey.
- Operational Definitions: Constructs like ''difficulty'' or ''proficiency'' may be subjectively interpreted across participants, posing potential validity threats. Mitigation: Clearly defined constructs were provided before the start of the experiment. The participants underwent a brief orientation to ensure a shared understanding of the terminologies and metrics.

In summary, while this study on the nMorph framework offers valuable initial insights, potential validity threats were actively recognized and addressed where possible. This marks an essential first step in a broader research journey, with plans for future studies to further substantiate and expand upon these findings, enriching the body of knowledge surrounding the nMorph framework.

V. CONCLUSION AND FUTURE WORK

A. CONCLUSION

In the rapidly evolving landscape of software development, the imperative of adaptability, scalability, and performance has never been more pronounced. The nMorph framework, drawing inspiration from the ''Transpiler-Based Architecture Design Model for Back-End Layers in Software Development,'' emerges as a potential solution to these challenges. This revolutionary framework allows developers to craft code in a single language, which can then be effortlessly transpiled into multiple target languages, ensuring that the core integrity of the original design remains uncompromised.

The nMorph framework boasts of a multifaceted architectural design tailored to robust enterprise back-end systems. By embracing transpilable languages, it ensures consistent performance regardless of the platform-specific output. It is complemented by a suite of specialized libraries and tools, with the Haxe transpiler taking center stage, ensuring seamless integration of the platform's web services, ORM functionalities, service exposure, and streamlined build process.

However, its real-world applicability, as revealed by an empirical experiment using throwaway prototypes, presents a more nuanced picture. When novice programmers were tasked with implementing business logic using the nMorph framework in Haxe and then comparing it to conventional languages, several salient points emerged. Notably, there was discernible variation in perceived task difficulty and the time taken across different development phases. On average, Haxe implementations took 20.11 minutes longer than conventional languages. Further delving into individual language dynamics, the nMorph framework presented heightened challenges for PHP users in terms of duration, and for C# users in terms of complexity. Another pivotal observation was the role of language proficiency, which had a marked influence during the analysis phase but saw its impact taper off during the development phase.

In concluding remarks, the nMorph framework, with its transformative potential, undeniably stands to reshape the approach to multi-platform software development. Its innovative methodology and a gamut of dedicated tools underscore its potential to redefine how developers navigate multi-language projects. However, empirical research offers a note of caution, highlighting the importance of understanding the intricacies of individual languages and their associated challenges. The pronounced learning curve, especially for those unfamiliar with the Haxe language and nMorph framework, underscores the pressing need for comprehensive documentation and robust training resources.

As the framework charts its future trajectory, the inclusion of intensive training modules and a broader spectrum of data-collection methodologies can significantly smoothen the transition between traditional languages and the nMorph paradigm.

To encapsulate this, the nMorph framework heralds a new era in software development, marrying innovation with practicality. However, its successful deployment mandates a deep-seated understanding of its capabilities, paired with focused training to unlock its myriad possibilities.

B. FUTURE WORK

The initial findings and insights derived from our research on the nMorph framework provide a promising foundation upon which we can build further studies. As we move forward, the following areas will be prioritized to bring the framework to its full potential and maximize its benefits for the broader software development community. Those are presented in priority order.

1) Real-World Case Studies: While empirical experiment with novice programmers provided valuable initial insights, it is imperative to test the framework's capabilities in real-world scenarios. We aim to collaborate with industry partners to implement nMorph in practical projects, and gauge its performance and adaptability in diverse settings. This will provide a more comprehensive view of the strengths and limitations of the framework, driving necessary refinements.
2) Broader Demographics: To ensure the wide applicability of the framework, our next steps involve replicating the study with different stakeholder groups. This includes professionals with varying experience levels, teams from different industries, and possibly integrating non-developers such as systems analysts or project managers. Their feedback and experience will help address the spectrum of challenges encountered in real-world software development.
3) Diverse Scenarios: The initial study centered on a banking transaction exercise. In future studies, we aim to explore other application domains, ranging from e-commerce and healthcare to entertainment and education. This will test the flexibility of nMorph in catering to the distinct requirements and constraints inherent to different sectors.
4) Framework Enhancement: As we collect more data and feedback, there is a continual evolution of the nMorph framework. Areas highlighted for improvement, such as automated testing, are prioritized. New tools and libraries may also be integrated based on emerging needs.
5) Comprehensive Documentation and Training: Based on the findings that underline the importance of robust documentation and training, efforts should be directed towards creating detailed user guides, tutorials, and workshops. These resources aim to facilitate a smoother transition for developers that adopt the nMorph framework.
6) Collaboration and Open-Source Initiatives: Considering the potential impact of the nMorph framework on multi-platform software development, we are contemplating open-source collaborations. This will allow developers worldwide to contribute to the framework, making it more robust and rich in features.

In conclusion, the nMorph framework's journey has just begun. Although the initial results are promising, there is an expansive roadmap ahead. By delving into real-world case studies, broadening our participant demographics, and refining the framework continually, we aim to use it in a transformative era of multi-platform software development.

REFERENCES

[1] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, 1st ed. Upper Saddle River, NJ, USA: Prentice-Hall, 1996.
[2] M. Fowler, *Patterns of Enterprise Application Architecture*. Reading, MA, USA: Addison-Wesley, 2012.
[3] T. Erl, *SOA Principles of Service Design* (The Prentice Hall Service-Oriented Computing Series From Thomas Erl). Upper Saddle River, NJ, USA: Prentice-Hall, 2007.
[4] S. Newman, *Building Microservices*. Sebastopol, CA, USA: O'Reilly Media, 2021.
[5] A. Bastidas Fuertes, M. Pérez, and J. Meza, "Transpiler-based architecture design model for back-end layers in software development," *Appl. Sci.*, vol. 13, no. 20, p. 11371, Oct. 2023.
[6] *System Requirements for Java*, Oracle, Austin, TX, USA, 2023. Accessed: Aug. 10, 2023.
[7] *.Net Core Release Notes*, Microsoft, Redmond, WA, USA, 2023. Accessed: Sep. 10, 2023.
[8] A. Bastidas Fuertes, M. Pérez, and J. Meza Hormaza, "Transpilers: A systematic mapping review of their usage in research and industry," *Appl. Sci.*, vol. 13, no. 6, p. 3667, Mar. 2023.
[9] Haxe Foundation. (2023). *Haxe—The Cross-Platform Toolkit*. Accessed: Aug. 10, 2023. [Online]. Available: https://haxe.org/
[10] Oracle. (2023). *Oracle Database*. Accessed: Aug. 10, 2023. [Online]. Available: https://www.oracle.com/database/
[11] Microsoft. (2023). *Microsoft SQL Server*. Accessed: Aug. 10, 2023. [Online]. Available: https://www.microsoft.com/en-us/sql-server/
[12] The PostgreSQL Global Development Group. (2023). *PostgreSQL: The World's Most Advanced Open Source Relational Database*. Accessed: Aug. 10, 2023. [Online]. Available: https://www.postgresql.org/
[13] MariaDB Foundation. (2023). *MariaDB: The Open Source Relational Database*. Accessed: Aug. 10, 2023. [Online]. Available: https://mariadb.org/
[14] Oracle Corporation. (2023). *MySQL: The World's Most Popular Open Source Database*. Accessed: Aug. 10, 2023. [Online]. Available: https://www.mysql.com/
[15] P. J. Leach, M. Mealling, and R. Salz, *A Universally unique Identifier (UUID) Urn Namespace*, document RFC 4122, Internet Engineering Task Force (IETF), Jul. 2005. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc4122
[16] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Berlin, Germany: Springer, 2012.
[17] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*. New York, NY, USA: Routledge, 2013.
[18] B. F. Andrés and M. Pérez, "Transpiler-based architecture for multi-platform web applications," in *Proc. IEEE 2nd Ecuador Tech. Chapters Meeting (ETCM)*, Oct. 2017, pp. 1–6.
[19] F. A. Bastidas and M. Pérez, "A systematic review on transpiler usage for transaction-oriented applications," in *Proc. IEEE 3rd Ecuador Tech. Chapters Meeting (ETCM)*, Oct. 2018, pp. 1–6.

[20] A. S. Kuznetsov, R. Y. Tsarev, T. N. Yamskikh, A. N. Knyazkov, A. N. Pupkov, and F. A. A. Laleye, "Compiler-compiler of multi syntax programming languages for creating N-version software," *J. Phys., Conf. Ser.*, vol. 1333, no. 7, Oct. 2019, Art. no. 072014.
[21] M. Kargar, A. Isazadeh, and H. Izadkhah, "Multi-programming language software systems modularization," *Comput. Electr. Eng.*, vol. 80, Dec. 2019, Art. no. 106500.
[22] Z. Anik and Ö. F. Baykoç, "Comparison of the most popular object-oriented software languages and criterions for introductory programming courses with analytic network process: A pilot study," *Comput. Appl. Eng. Educ.*, vol. 19, no. 1, pp. 89–96, Mar. 2011.

ANDRÉS BASTIDAS FUERTES (Member, IEEE) received the degree in computer system engineering and the dual master's degree in strategic software management and high enterprise management. He is currently pursuing the Ph.D. degree in computer science with a specialization in software engineering with Escuela Politécnica Nacional University. He is the CEO of Smartwork S.A. and an ISO Auditor. He possesses international MCPD, MCP, MCTS, and other technical certifications, as well as a Six Sigma Green Belt certification and SCRUM Master. He specializes in the insurance and financial industries. He is also a consultant on multiple digital entrepreneurship platforms and a software architect, conceptualizer, designer, and implementer of software solutions, as well as a designer and implementer of business processes, technology strategy design, digital transformation mentors, and university teachers. He has more than 20 years of experience in software development and technological implementation in the industry. He has participated in more than 200 comprehensive IT solutions in both private and public sectors.

MARÍA PÉREZ received the Ph.D. degree in computer science and mathematical modeling. She is currently an Assistant Professor with the Department of Computer Science and Informatics, Escuela Politécnica Nacional, Quito, Ecuador. She has participated in several projects, such as computer vision, pattern recognition, digital images, and information systems, and several publications in journals with impact factors. Her research interests include artificial intelligence and vision, in particular, computer methods for early diagnosis and classification of cancerous lesions. She has been a reviewer of several international congresses and journals. Among the other roles, she performs.

JAIME MEZA (Member, IEEE) received the master's degree in business administration and computer engineering and the Ph.D. degree (cum laude) in project and systems engineering from the Polytechnic University of Catalonia, Spain. He is currently a Postdoctoral Researcher with the University of Fribourg, Switzerland; a Full Professor with Universidad Técnica de Manabí, Ecuador; and a Staff Member of IEEE e-government STC. He has been a teacher for more than 15 years with multiple universities in Ecuador as well as an academic guest with the Polytechnic University of Catalonia, Spain, and the University of Freiburg, Switzerland. In the professional field, he has held various management positions, such as the project manager, the chair, an IT advisor, and an auditor, at the local and national levels in companies and government institutions. His research projects seek to reduce tax evasion/fraud, improve tax collection using AI systems and advanced analytical techniques, and collaborative spatial urban planning supported by cognitive and recommendation systems. His research interests include collective intelligence, soft computing, recommender systems, and models of collaborative cognition to improve public services and higher education.