



Project

G.A.M.E.D.E.V.

Concurrent IO

Software Requirements Specification

Contents

1	Introduction	2
1.1	Context	2
1.2	Product Description	3
1.3	Intended Use	3
1.4	Limitations	3
1.5	Target Groups	3
2	Specification	4
2.1	Functional Requirements	4
2.2	Non-functional Requirements	6
2.2.1	Interface Requirements	6
2.2.2	Performance Requirements	7
2.2.3	Usability	7
2.2.4	Other Non-Functional Requirements	8
3	System	11
3.1	Build	11
3.2	Continuous Integration	11
3.3	Version Control	11
3.4	Code Quality	11
3.5	Logging	11
4	Tests	12
4.1	Test Plan	12
4.2	Unit Tests	12
4.3	Integration Tests	12
4.4	System Tests	12
4.5	Performance Tests	13
4.6	Installation Tests	13

Chapter 1

Introduction

1.1 Context

Almost every game or game development tool requires some way of accessing files or directories on the underlying system, most often with the purpose of saving the game state, or for the purposes of logging. The C# language provides many ways to perform I/O operations[8], bringing additional complexity to the problem of implementing I/O. As of May 2025, there are over 2000 results on the Unity Asset Store[7] when searching for tools marked with the keyword "file"[1]. Yet most of these results provide project-specific functionality, and few of the results are built with the idea of abstracting away the complexity of dealing with C#'s I/O classes directly. Furthermore, little is done to ensure software industry standards are kept while developing those tools. Most of the aforementioned assets lack thread-safety, or are not optimized to work concurrently with many files and directories. There is little to no transparency of the underlying implementations (assets using the `File` class[9] vs e.g. the `StreamReader` class[10]), with each system employing different methods. Rarely is there industry-standard quality assurance such as continuous integration, automated tests, and fault protocols in case of unforeseen system circumstances (e.g. system shutdown while managing files, memory limitations, code error or exception during I/O...). Often there is lack of proper coding standards while designing and implementing systems working with files (race conditions, deadlocks, unreleased memory buffers...).

For these reasons, a file and directory (I/O) management system is needed that follows industry-standard quality assurance, is thoroughly tested and documented, and has transparent requirements.

1.2 Product Description

The purpose of the I/O manager system is to provide a thread-safe and optimized wrapper utility for managing files and directories from Unity scripts.

The system consists of C# scripts that can be called from the user's own scripts like a library. No attached in-game components are necessary. The system's scripts are bundled in an assembly definition file that can be referenced by the user's own assembly definition files in order for the system's namespace to be accessible.

The system shall provide thread-safe wrapper methods for all supported methods of the .NET classes `File`[11] and `Directory`[12] to allow concurrent I/O when using this I/O utility. A supported method is a method which:

- is supported on the .NET version used by Unity,
- is supported on all operating systems specified below,
- and can be reasonably implemented in a thread-safe manner.

1.3 Intended Use

The system is intended to be used as a software library inside of a Unity project, and is not meant to be used as a standalone software solution. The system is meant to be interacted with from user scripts only.

1.4 Limitations

The system shall only support the operating systems that support I/O using the `File` and `Directory` classes. The following operating systems shall be explicitly tested: Standalone Windows/Linux/OSX, Android, iOS.

The system shall serve as a lightweight thread-safe wrapper for the I/O methods it invokes. There shall be only limited exception handling, parameter checking, or additional logic. Further operating system support or features may be added in later versions of the system at the developer's discretion.

1.5 Target Groups

The system is targeted at game or tool developers using Unity whose games or tools require single-threaded or concurrent I/O access at any point during gameplay. The system is also planned to be used internally by further Project G.A.M.E.D.E.V. tools as a means of safely and efficiently performing concurrent I/O.

Chapter 2

Specification

2.1 Functional Requirements

ID: 1	Name: File methods (supported)
• The system implements the following methods (and their overrides) from the <code>File</code> class in a thread-safe manner: <code>AppendAllLines</code>, <code>AppendAllText</code>, <code>Copy</code>, <code>Create</code>, <code>Delete</code>, <code>Exists</code>, <code>GetAttributes</code>, <code>GetCreationTime</code>, <code>GetCreationTimeUtc</code>, <code>GetLastAccessTime</code>, <code>GetLastAccessTimeUtc</code>, <code>GetLastWriteTime</code>, <code>GetLastWriteTimeUtc</code>, <code>Move</code>, <code>ReadAllBytes</code>, <code>ReadAllLines</code>, <code>ReadAllText</code>, <code>Replace</code>, <code>SetAttributes</code>, <code>SetCreationTime</code>, <code>SetCreationTimeUtc</code>, <code>SetLastAccessTime</code>, <code>SetLastAccessTimeUtc</code>, <code>SetLastWriteTime</code>, <code>SetLastWriteTimeUtc</code>, <code>WriteAllBytes</code>, <code>WriteAllLines</code>, <code>WriteAllText</code>.	

ID: 2	Name: File methods (not supported)
The system does not implement the following methods: • Decrypt and Encrypt – these methods are Windows-only and are no longer recommended for use; moreover, they have better alternatives in the Unity API. • GetUnixFileMode and SetUnixFileMode – these methods are not supported on Windows. • OpenHandle – this method allows non-mutex multi-threaded access to files, which is incompatible with this system. • Move(String, String, Boolean) and ResolveLinkTarget – these methods are not available on the .NET version supported by Unity. • SetCreationTime(SafeFileHandle, DateTime), SetCreationTimeUtc(SafeFileHandle, DateTime), SetLastAccessTime(SafeFileHandle, DateTime), SetLastAccessTimeUtc(SafeFileHandle, DateTime), SetLastWriteTime(SafeFileHandle, DateTime), and SetLastWriteTimeUtc(SafeFileHandle, DateTime) – those overloads are not available on the .NET version supported by Unity. • AppendAllLinesAsync, AppendAllTextAsync, ReadAllBytesAsync, ReadAllLinesAsync, ReadAllTextAsync, ReadLinesAsync, WriteAllBytesAsync, WriteAllLinesAsync, and WriteAllTextAsync – asynchronous methods are not compatible with the system. • AppendText, CreateSymbolicLink, CreateText, Open, OpenRead, OpenText, OpenWrite – these methods return objects that allow external (not within the system) modification of the path. • Methods that take SafeFileHandle instead of a string path as a parameter (e.g. GetAttributes) – the string path necessary for the system's internal concurrency mechanism cannot be extracted from a SafeFileHandle object. • ReadLines – This method cannot be supported by the system, as it returns an enumerator for the file contents, meaning no other thread can access the file at the same time; this enumerator is outside of the scope of the system and with it, file contents can be accessed at a later time.	

ID: 3	Name: Directory methods (supported)
The system implements the following methods (and their overrides) from the <code>Directory</code> class in a thread-safe manner: <code>CreateDirectory</code>, <code>Delete</code>, <code>EnumerateDirectories</code>, <code>EnumerateFiles</code>, <code>EnumerateFileSystemEntries</code>, <code>Exists</code>, <code>GetCreationTime</code>, <code>GetCreationTimeUtc</code>, <code>GetCurrentDirectory</code>, <code>GetDirectories</code>, <code>GetDirectoryRoot</code>, <code>GetFiles</code>, <code>GetFileSystemEntries</code>, <code>GetLastAccessTime</code>, <code>GetLastAccessTimeUtc</code>, <code>GetLastWriteTime</code>, <code>GetLastWriteTimeUtc</code>, <code>GetLogicalDrives</code>, <code>GetParent</code>, <code>Move</code>, <code>SetCreationTime</code>, <code>SetCreationTimeUtc</code>, <code>SetCurrentDirectory</code>, <code>SetLastAccessTime</code>, <code>SetLastAccessTimeUtc</code>, <code>SetLastWriteTime</code>, <code>SetLastWriteTimeUtc</code>.	
ID: 4	Name: Directory methods (not supported)
The system does not implement the following methods: <code>CreateTempSubdirectory</code> – this method is not available on the .NET version supported by Unity. <code>CreateDirectory(String, UnixFileMode)</code> – <code>UnixFileMode</code> is not available on the .NET version supported by Unity. <code>CreateSymbolicLink</code>, <code>ResolveLinkTarget</code> – these methods return objects that allow external (not within the system) modification of the path.	

2.2 Non-functional Requirements

2.2.1 Interface Requirements

ID: 5	Name: Code library
- The system is a code library, and as such can be called directly from user scripts. - The system is not attached to any scene components, but is used directly from the user's scripts.	

2.2.2 Performance Requirements

ID: 6	Name: Concurrent I/O optimization
• The system is optimized for working with many files and directories simultaneously. • The system does not use a mutual exclusion lock between all methods, but allows many files to be accessed concurrently via named locks. • Mutual exclusion locking is allowed for directory access.	

ID: 7	Name: High concurrency reliability
• The system is expected to perform IO operations safely, regardless of thread count or underlying OS speed. • A thread count of 1024 on a high-performance machine (high-core, high-speed CPU, high-speed SSD) is to be explicitly tested.	

2.2.3 Usability

ID: 8	Name: System access
• The scripts of the system are part of an assembly definition file that has to be referenced from the user's own assemblies to access the system's namespace.	

ID: 9	Name: Default settings
• The system has reasonable default settings for its options.	

ID: 10	Name: Informing the user
• The system throws specific exceptions in order to inform the user of an unsafe state, or an invalid operation or arguments.	

2.2.4 Other Non-Functional Requirements

Reliability

ID: 11	Name: IO reliability
• The system uses the provided System.IO methods in order to guarantee no IO errors are introduced.	

ID: 12	Name: Crashing
• The system never crashes during normal execution or while in a safe state.	

ID: 13	Name: Unsafe state
• The system throws an exception upon entering an unsafe state or undefined behavior, e.g. calling methods with empty parameters. This ensures the system never continues operating in such states.	

ID: 14	Name: Error handling
• The system never crashes under normal execution. • The system does not handle any exceptions caused by the underlying methods.	

Portability

ID: 15	Name: Supported operating systems
• The system only supports the operating systems that support I/O using the <code>File</code> and <code>Directory</code> classes. The following operating systems are explicitly tested: Standalone Windows/Linux/OSX, Android, iOS. • Further operating systems supported by Unity and System.IO may work, but are not explicitly tested.	

ID: 16	Name: Compatibility
• The system is compatible with all other Project G.A.M.E.D.E.V. tools by sharing the same folder and namespace structure. • All scripts of the system are contained in custom assembly definitions, separating them from the rest of the (Unity and user) scripts in the project.	

Scalability

ID: 17	Name: Supporting additional I/O functionality
• The system is modeled in such a way as to easily allow support for additional I/O classes and methods.	

Re-usability

ID: 18	Name: Internal Usage
• The system is used by all further Project G.A.M.E.D.E.V. tools for safe and reliable concurrent IO operations.	

Compatibility

ID: 19	Name: Unity editor support
• The system supports all stable Unity versions starting from the lowest allowed version for publishing an asset on the Unity asset store. • The system can be used with any render pipeline. • The system uses .NET standard 2.1 (supported since Unity 2021.3).	

Localization

ID: 20	Name: Language
• The system does not support localization. • All source code and documents are in English.	

Chapter 3

System

3.1 Build

The system is bundled together with the built Unity scene(s). As per its intended use, the system cannot be built as standalone software, and is only usable inside of a Unity build.

3.2 Continuous Integration

Upon each push to the remote repository, the continuous integration pipeline is started by a GitLab^[2] runner. Automated tests are all run automatically by the pipeline. Separate jobs are performed for each supported operating system using Docker images. Due to technical limitations, only some of the supported Unity versions are tested by the pipeline, but at least one of them is a LTS version.

To assist with the continuous integration, CircleCI's Unity Orb^[3] under GitLab is used.

3.3 Version Control

For this project, Git using GitLab as a remote location is selected as a version control system.

3.4 Code Quality

C# coding conventions and best practices are enforced throughout the source code. SonarLint^[4] is used locally to assist with static analysis of the source code.

3.5 Logging

Logging is not planned for this project.

Chapter 4

Tests

4.1 Test Plan

Each requirement in this document shall be checked by automated tests. Requirements that cannot be checked by automatisisation (e.g. some non-functional requirements) shall be manually tested.

Where applicable, fuzzing shall be employed for providing tests with non-repeating, dynamic values (e.g. random strings, objects with dynamic content), and stubbing/mocking shall be used for simulating specific system states (most useful during performance testing).

Persistent test reports, BDD (behavior-driven development), TDD (test-driven development), and equivalence class analysis are not planned for this project.

Due to the small scale of the system, above-industry coverage standard shall be applied to the project. Testing shall be considered sufficient at 100% requirement and 100% method coverage, and at least 70% statement coverage.

4.2 Unit Tests

The purpose of unit tests shall be to test all modules individually. Unit tests shall be written after each module creation.

4.3 Integration Tests

The purpose of integration tests shall be to test the correctness of individual modules working together. Only tested modules, i.e. those with unit tests, shall be used for integration testing. Similar to the unit tests, integration tests shall be created continuously during development.

4.4 System Tests

The purpose of system tests shall be to test the correctness of the entire system. They are similar to integration tests, but test the entirety of the system instead of specific integrated modules. System tests shall be created after all functionality is present and tested by unit and integration tests.

4.5 Performance Tests

Sometimes also called load tests, the purpose of performance tests shall be to test the system's performance (speed) and reliability under extreme inputs. Performance tests shall be created after the system tests.

4.6 Installation Tests

The purpose of installation tests shall be to verify the system's integrity on scene start. This includes, but is not limited to, necessary files present and intact, current OS supported, main component attached to an active game object etc. Installation tests shall be created as a last step of the development process, after the performance tests and before system deployment.

Bibliography

- [1] Unity Asset Store. Tools with keyword "file".
<https://assetstore.unity.com/?category=tools&q=file&>
- [2] GitLab. <https://about.gitlab.com/>
- [3] CircleCI Developer. Unity Orb.
<https://circleci.com/developer/orbs/orb/game-ci/unity>
- [4] SonarSource. SonarLint.
<https://plugins.jetbrains.com/plugin/7973-sonarqube-for-ide>
- [5] Project G.A.M.E.D.E.V. Tools, Shaders, Games. www.project-gamedev.com
- [6] Project G.A.M.E.D.E.V. Documentation. www.docs.project-gamedev.com
- [7] Unity Asset Store. <https://assetstore.unity.com/>
- [8] Microsoft. Use Visual C# to do basic file I/O.
<https://learn.microsoft.com/en-us/troubleshoot/developer/visualstudio/csharp/language-compilers/file-io-operation>
- [9] Microsoft. File Class.
<https://learn.microsoft.com/en-us/dotnet/api/system.io.file>
- [10] Microsoft. StreamReader Class.
<https://learn.microsoft.com/en-us/dotnet/api/system.io.streamreader>
- [11] Microsoft. File Class. Methods.
<https://learn.microsoft.com/en-us/dotnet/api/system.io.file#methods>
- [12] Microsoft. Directory Class. Methods. <https://learn.microsoft.com/en-us/dotnet/api/system.io.directory#methods>