Software Requirements Specification (SRS)

by TophUwO

June 2024

Project **Noriko**

Component **Noriko**

Noriko is a cross-platform 2-D RPG game engine, focused on simplicity, ease of use, and scalability.

Table of Contents

Version History

The following table logs all changes made to this document during the analysis phase of the project. Author names may hold hyperlinks to the author's preferred way of contact. Dates are given in ISO-8601 **`MM-DD-YYY`** format. Version numbering is as follows: **`MAJOR.MINOR.PATCH[-ITERATION]`**. Version **`1.0.0`** marks the release version of this document according to which the software will be developed.

DATE	VERSION	CONTRIBUTOR	CHANGES
06-16-24	0.0.1-1	TophUwO	**INITIAL COMMIT** **ADD** table of contents
06-16-24	0.0.1-2	TophUwO	**ADD** version history **DO** accent color setup
06-16-24	0.0.1-3	TophUwO	**ADD** remaining major sections **FIX** formatting of tables
06-17-24	0.0.2-1	TophUwO	**ADD** initial content for all sub-sections within section *1. Introduction*
06-17-24	0.0.2-2	TophUwO	**ADD** text contents for section *2.1. Product Perspective and System Context*
06-17-24	0.0.2-3	TophUwO	**ADD** CTXD for Noriko's ecosystem **ADD** remaining text for section *2.1. Product Perspective and System Context*
06-17-24	0.0.2-4	TophUwO	**ADD** initial content for section *2.3. User Classes and Characteristics* **ADD** content for section *2.5. Design and Implementation Constraints*
06-18-24	0.0.3	TophUwO	**ADD** contents for section *2.6. Coding Style*
06-18-24	0.0.4	TophUwO	**ADD** contents for section *2.4. Operating Environment*
06-18-24	0.0.5	TophUwO	**ADD** product feature overview
06-18-24	0.1.0-1	TophUwO	**ADD** contents for sections *2.7. User Documentation* and *2.8. Assumptions and Dependencies*
06-18-24	0.1.0-2	TophUwO	**FIX** dependency direction error in **`CTXD`**
06-19-24	0.1.0-3	TophUwO	**EXPAND** remaining sections
06-19-24	0.2.0	TophUwO	**ADD** system features (*4. System Features*)
06-19-24	0.2.5	TophUwO	**ADD** ISO/IEC 25010:2011 software quality attribute table + priority ratings
06-19-24	0.2.6	TophUwO	**ADD** explanations for debugging tools **ADD** debug screen requirements
06-20-24	0.2.7	TophUwO	**ADD** contents for section *6.1.2. Debug Console*
06-20-24	0.2.8	TophUwO	**ADD** contents for section *6.1.3. Logs*
06-21-24	0.2.9	TophUwO	**ADD** contents for section *6.1.4. Crash Dumps*
06-21-24	0.3.0-1	TophUwO	**ADD** initial version of package diagram (**`PKGD`**)
06-21-24	0.3.0-2	TophUwO	**UPDATE** **`PKGD`**

06-21-24	0.3.0-3	TophUwO	**FIX** circles in **PKGD**
06-21-24	0.3.0-4	TophUwO	**FIX** formatting in **PKGD**
06-21-24	0.3.1	TophUwO	**ADD** description for **PKGD**
06-21-24	0.3.2	TophUwO	**ADD** main deployment diagram (**DPD**)
06-21-24	0.4.0-1	TophUwO	**UPDATE DPD** **ADD** contents for section *3. External Interface Requirements*
06-21-24	0.4.0-2	TophUwO	**DO** minor cleanups and typo fixes
06-22-24	0.5.0	TophUwO	**ADD** original user story
06-22-24	0.6.0	TophUwO	**REMOVE** empty and/or unused sections **DO** minor cleanups and typo fixes **FIX** some formatting issues
06-22-24	0.7.0	TophUwO	**ADD** OOA class diagram (**CD**)
06-22-24	0.7.1	TophUwO	**ADD** description for OOA CD
06-22-24	1.0.0	TophUwO	**INITIAL RELEASE VERSION**

1. Introduction

The following section provides basic information on not only the product described by this document, but also the document itself, such as document conventions, structure, and various pointers for readers.

1.1. Purpose

This document is a Software Requirements Specification (SRS) for the component called **`Noriko`** of project **`Noriko`**. Project Noriko is a cross-platform 2-D role-playing game (RPG) game engine. Other documents of this type exist for components **`NorikoEd`** and **`NorikoRt`**. Throughout the remainder of this document, the term **`Noriko`** will refer to the game engine component only.

1.2. Document Conventions

The document at its core closely adheres to IEEE 830: Software Requirements Specification. Additional sections providing miscellaneous information may exist, especially in the latter half of this document. The document's language is English (United States). Terms which are **`highlighted like this`** may be either technical terms, variables, or other information of special significance. In the Version History table, terms in **`CAPITAL BOLDFACE`** signify commit message actions, whereas terms in *`underlined highlighted italics`* are links to other sections within this document. When external links are inserted, a superscript `(»)` is appended to the link texts, whereas e-mail or other contact links are annotated via a superscript `(@)`.

1.3. Intended Audience and Reading Suggestions

The document is to be understood by all stakeholders. That includes project managers, designers, software engineers, and programmers. What sections are most important depends on the individual's function in the project. While project managers should be familiar with the entire document, software engineers and designers should focus on the latter half of this document which delves into low-level details of the system.

1.4. Product Scope

Games have seen a rapid increase in popularity in recent decades. With a milliard of genres to choose from, effortlessly crossing country-, demographic-, as well as cultural boundaries, games have become a vital part of today's entertainment industry. Whereas today's focus of AAA studios is mostly on realistic 3-D worlds, there was a time not long ago when computers and especially consoles lacked the hardware power to simulate scenery this complex. In the 1990s and early 2000s, handheld consoles became increasingly prevalent, with the Nintendo® Game Boy® being introduced as far back as 1989. Due to the stringent hardware limitations of early-generation handheld devices, games had to not only heavily optimize game- and drawing routines, but also had to place strict requirements on complexity as well as code size.

Having grown up in the early 2000s, I came in contact with handheld consoles, with the Nintendo® Game Boy Color® being my first one. I quickly developed a deep interest in RPG games which were virtually always set in tile-based worlds due to the aforementioned limitations. This project, Noriko, aims to recapture the iconic charm that managed and still manages to captivate millions of kids across the globe by focusing on what made these games special, with the objective of making development of such games as easy, accessible, efficient, and scalable as possible.

1.5. References

The following table contains resources (web pages, books, articles, ...) to key topics and technologies mentioned throughout this document. They may serve as an introductory pointer to the reader; however, sticking to them is not required. Additional research is advised.

TERM	TYPE	DATE	LINK
IEEE 830	website	06-19-24	*Click to jump!*(»)
ISO/IEC 9899:2011	PDF	06-19-24	*Click to jump!*(»)
TILE-BASED GAME	article	06-22-24	*Click to jump!*(»)

1.6. Applications and Tools Used

This section lists all tools and external applications used for creating this document, grouped by scope.

SCOPE	TOOL(S) USED
LAYOUT & TYPESETTING	Microsoft® Word® 365 (Version: 2405)
OOA DIAGRAMS	Eclipse® Papyrus (2023-12 (4.30))
CODE SNIPPETS	Microsoft® Visual Studio® 2022 Community (Version 17.8.4)

2. General Description

This section focuses on key aspects of the system, such as high-level features, user-, and system context analysis.

2.1. Product Perspective and System Context

All three main components of Noriko live in the same ecosystem and are designed to depend on one another in order to function properly. The ecosystem is largely self-sufficient; dependencies from outside the ecosystem are non-existent or minimal.

The following system context diagram visualizes the relationships between Noriko's three main components. Optional and external components are marked as such. Optional components may extend the system's functionality but are not required to exist for the system to provide all necessary features.

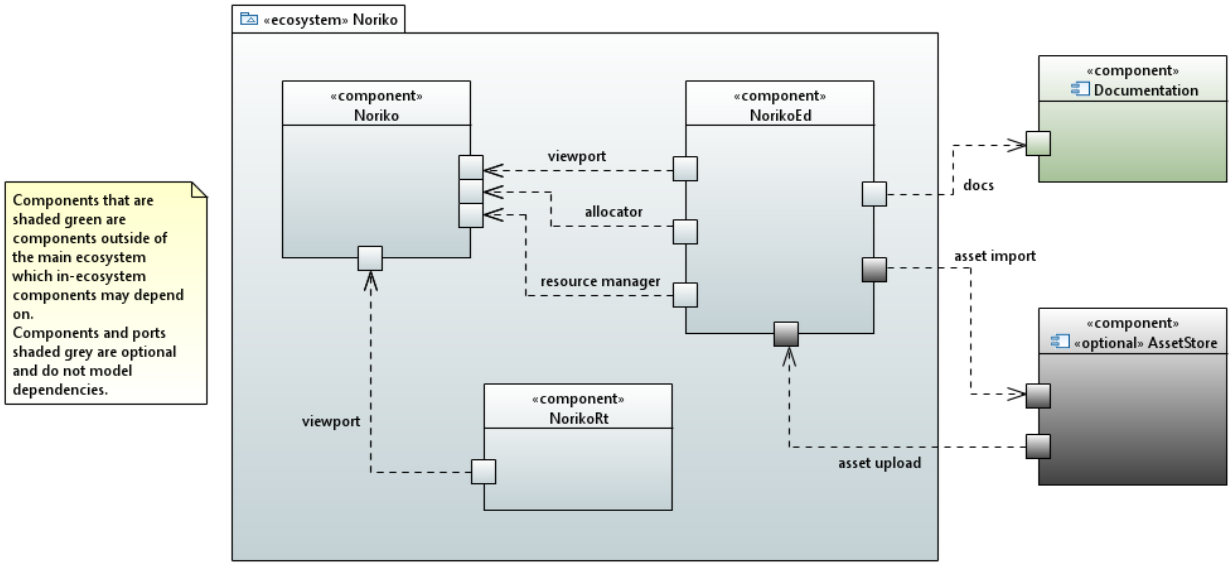

Figure 1: system context diagram (CTXD) showing Noriko's main three components as well as major dependencies.

This document serves as the requirements specification for the game engine component of Noriko. The components **`NorikoEd`** and **`NorikoRt`** are explained in detail in other documents.

2.2. Product Functions

Noriko is a 2-D RPG game engine ecosystem, featuring an engine component with the same name, **`NorikoEd`**, the in-ecosystem editor implementing the content creation pipeline, and **`NorikoRt`**, the runtime directly instantiating the engine component. In the following, only the engine component's core features are outlined:

- render 2-dimensional tile-based worlds and dynamic entities
- tiles can contain interactive event data
- support for animations
- support for advanced image effects (alpha blending, …)
- multiple UI overlays
- customizable GUI
- extensible debugging features (also graphical)
- scripting for events, dialogue, and entities
- multiple rendering technologies (APIs) supported
- particle system
- shader support
- networking
- audio support

2.3. User Classes and Characteristics

Noriko's does not feature a user hierarchy. The user is the developer or player of the game that was created using Noriko's tools.

2.4. Operating Environment

Noriko is intended to be run on desktop as well as embedded platforms. There are generally no minimum system requirements, except that the platform needs to have a graphic output raster device (i.e., a display). The only requirement is that a C11-conforming C-standard library is available on the target platform. Features that are infeasible to implement on less powerful platforms are optional. What features are infeasible must be determined during the implementation and testing phase.

For desktop and mobile platforms, here are the minimum system requirements that must support all product features outlined in this document:

PLATFORM	MINIMUM OPERATING SYSTEM VERSION
MICROSOFT WINDOWS	Microsoft Windows® 7 (Build: 7601)
LINUX	Linux® Kernel Version 3.2
MACOS	Apple macOS® 10.10.0 'Yosemite'
NINTENDO SWITCH	Nintendo® Switch® Firmware v16.0.0

Support for platforms not listed in the table above is optional.

2.5. Design and Implementation Constraints

The game engine is to be implemented in ISO C11 (ISO/IEC 9899:2011) where possible. Using APIs (application programming interfaces) which are written in a language other than C and/or exposed to the user via non-C interfaces is permitted but appropriate wrappers must be provided. There is no limitation as to what external libraries are allowed to be used, as long as they are released under a permissive software license such as the *`BSD 3-clause License`*(») or the *`Apache 2.0 License`*(»). As for the source code and inline documentation, the required language is **`English (United States)`**.

2.6. Coding Style

This section describes the coding style used in Noriko's codebase. This coding style is to ensure consistent quality, optimized for easy-to-read code and a variety of display hardware.

2.6.1. Rules

The following table categorizes language features into **`elements`** and lists the style rules that are valid for the given element. Note that the given character limits exclude any attached prefixes.

ELEMENT	RULE
FILE NAMES	○ not longer than 24 characters ○ all lower-case ○ contain only alphanumeric characters and _ 'underscore'
GLOBAL FUNCTION NAMES	○ not longer than 64 characters ○ start with **`Nk`** prefix ○ use camel-case
GLOBAL IDENTIFIERS	○ not longer than 32 characters ○ prefixed with **`gl_`** ○ use camel-case

GLOBAL CONSTANTS	○ see row **`GLOBAL IDENTIFIERS`** ○ infix **`_c_`** added between **`gl_`** and the variable name
STRUCTURE NAMES AND TAGS	○ see row **`GLOBAL FUNCTION NAMES`**
VARIABLES DECLARED IN STRUCTS	○ not longer than 16 characters ○ prefixed with **`m_`** (normal) or **`mp_`** (pointers) ○ use camel-case
FUNCTION POINTERS DECLARED IN STRUCTS	○ not longer than 16 characters ○ use camel-case
LOCAL VARIABLES	○ see row **`FUNCTION POINTERS DECLARED IN STRUCTS`**
LABELS	○ not longer than 16 characters ○ all upper-case ○ prefixed with **`lbl_`**
ENUM CONSTANTS	○ use PascalCase for identifier ○ prefix with user-defined scope identifier, followed by a **_** 'underscore', indicating the enum the constant belongs to ○ use PascalCase for prefix as well
MACROS	○ not longer than 32 characters ○ all upper-case ○ use **`NK_`** prefix

2.6.2. Formatting

The following formatting rules apply:

1) single lines not longer than 120 characters
2) opening **`{`** 'brace' are placed on the same line as the identifier or control block
3) control blocks with one statement do not use **`{}`** 'braces'
4) indirection **`*`** 'asterisk' attach to the identifier instead of the type
5) **`else (if)`** are placed on the same line as the closing **`}`** 'brace' or on the line after the statement if **`}`** are omitted due to rule **`3`**
6) use 4 (four) spaces for tab-stops (NEVER use actual tabs)
7) nested preprocessor directives are indented just like code
8) add one space after control blocks and between control block expression and opening **`{`** 'brace'
9) do not use parenthesis for **`return`**, **`sizeof`** and alike (unless syntactically and/or semantically required)
10) insert space after commas in argument and initializer lists
11) always use **`/* */`**-style comment blocks

12) use any valid DOXYGEN-style comments for inline documentation
13) add spaces between (binary) mathematical operators and their operands
14) macro values are always enclosed in **()** 'parentheses'
15) in a declaration, annotations come first, then storage classes, then the type, and then type qualifiers
16) grouped declarations that span multiple lines are aligned so that their identifiers start on the same column
17) labels are always placed one level of indentation to the left of the current level in their own separate line
18) use blank lines liberally – within function, group code based on what it contributes to
19) in implementation (***.c**) files, static functions come before exported functions
20) symbol order within header and implementation files:
    a) `#include`
    b) `#define`
    c) `struct`/`enum`/`typedef` declarations & definitions
    d) `static` functions
    e) exported functions

2.6.3. Full Example

The following code showcases the coding style described in the above sections. Lines may have a comment such as **/* 8. (& ...) */**, indicating what formatting rules were applied.

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>



/* Demonstrating nested preprocessor directives. */        /* 11. */
#if (defined NK_LEVEL_1_PREPROC)
    #if (defined NK_LEVEL_2_PREPROC)                      /* 7. */
        #define NK_LEVEL (3)                              /* 14. */

        #include <windows.h>
    #elif (defined _DEBUG)
        #error Cannot compile in debug mode.
    #endif
#endif
```

```c
#define NK_ANNOTATION

/* Demonstrating global variables. */
NK_ANNOTATION static int const gl_c_displayWidth = 1080;   /* 15. */

/* Demonstrating types. */
typedef int NkErrorCode;

/**
 * \brief This is a DOXYGEN-style comment.
 *
 * This is the detailed description for this declaration.
 *
 * \note  This is a note.
 */
typedef struct NkStaticApplicationContext {                /* 2. */
    uint32_t  m_uintVal;                                   /* 16. */
    uint32_t *mp_uintPtrVal;                               /* 4. & 16. */

    void (__cdecl *getObject)(char const *const id);       /* 4. & 15. */
};
/* Demonstrating enumerations. */
typedef enum NkErrorCode {                                 /* 2. */
    NkErr_Ok,
    NkErr_Unknown,

    NkErr_NotImplemented,
    NkErr_OutOfMemory
} NkErrorCode;


/* Demonstrating functions. */
NK_ANNOTATION inline NkErrorCode NkCreateWindow(           /* 1. & 15. */
    uint32_t width,
    uint32_t height,
    uint8_t bbp
) {
    /* Validate parameters. */
    if (!width || !height)                                 /* 3. & 8. */
        return NkErr_NotImplemented;

    /* Do stuff. */
    void *ptr = calloc(1, sizeof width * 32);              /* 4. & 9. & 10. & 13. */
    if (!ptr)
        return NkErr_OutOfMemory;
    else if (ptr == 0x01) {                                /* 5. */
        printf("Pointer was 0x01.\n");

        goto lbl_ERR;                                      /* 9. */
    }

    /* Clean up and return. */
    free(ptr);                                             /* 18. */
    return NkErr_Ok;
```

```
    /* Clean up in case of error. */
lbl_ERR:                                                    /* 17. */
    free(ptr);
    return NkErr_Unknown;
}
```

2.7. User Documentation

Since the engine itself is not used by players directly, its user documentation is to be provided in the form of inline documentation, generated with the help of **DOXYGEN**, and tutorials.

2.8. Assumptions and Dependencies

For the engine to function properly, a readable and writable filesystem must be present on the side of the target platform as well as a way to manually manage memory. Furthermore, the engine depends on the C-standard library for various core functions, so it either must be present on the host platform or deployed alongside the engine.

3. External Interface Requirements

3.1. User Interfaces

As Noriko itself is implemented as a shared library, the user interface is the engine's public API, invoked by both the **`NorikoEd`** and **`NorikoRt`** components. For details on user interfaces for these components, refer to their requirements specification.

3.2. Hardware Interfaces

Noriko's engine component, being a graphical real-time interactive application, with various types of hardware, especially GPUs and HI (human interface) devices. Noriko implements multiple GPU interfaces, using various graphics APIs such as **`OpenGL`**, **`Vulkan`** and platform-specific technology. The API that will be used for a given session is chosen by the application automatically (default) or can be selected by the user upon engine startup. What API is chosen automatically depends on availability and suitability. For this purpose, Noriko determines the suitability of a given graphics technology based on the features the game requires. Feature requirements are generated when the game is deployed.

For displaying, Noriko chooses the main display defined by the host platform. If there is no display connected to the host platform, Noriko will not run. On window-based host platforms, Noriko is designed to run in windowed mode by default but also supports full screen.

Regarding input devices, Noriko must support keyboard (and mouse) input as well as joy-stick input. For this purpose, Noriko must support mapping arbitrary inputs specific to the current input device to generic actions defined by the engine. Such a generic action could be **`Open Menu`** or **`Move One Tile Northwards`**. There must be a for the game to define new actions and supply mappings. Built-in and custom mappings can be overridden by each game individually.

3.3. Software Interfaces

The following deployment diagram illustrates the execution architecture of Noriko. Artifacts shaded yellow/orange are part of the engine component.

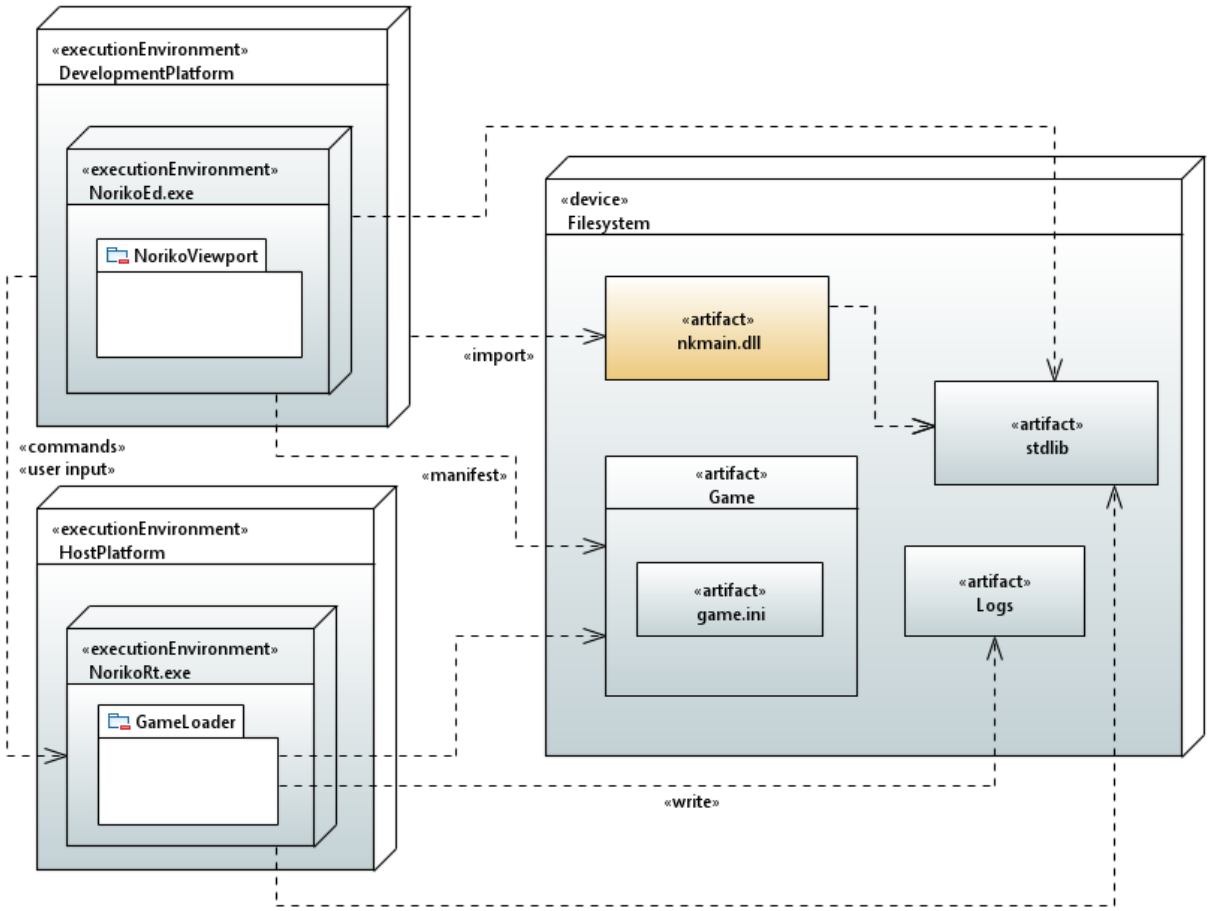

Figure 2: Noriko's execution/deployment architecture

After the **`NorikoEd`** component, the engine's editor and content creation pipeline implementation, has created the game files and specifically the environment information exposed by game.ini, the **`NorikoRt`** (runtime) component can open this file and start the game. **`NorikoRt`** can also be invoked by the **`NorikoEd`** component via a viewport, directing user input and commands to the runtime component.

Note: the above deployment diagram shows Windows file extensions. Other platforms use their own equivalent, if present.

3.4. Communications Interfaces

Communication between Noriko other software in the Noriko ecosystem is carried out through Noriko's public API only. Other software in the ecosystem may have different communication requirements and may employ other means. For more details, refer to the requirements specification of the software in question.

4. System Features

The following section itemizes Noriko's main features, dependencies, and detailed descriptions for each feature. The requirements list is not in a particular order.

REQ-N	BRIEF	DEP	DESCRIPTION
REQ-1	input device support	-	○ support keyboard, mouse (and touch), and joy-stick input ○ provide mappings for each supported input device type ○ allow changing default mappings in special menus
REQ-2	multiple graphics APIs	platform	○ GDI (windows-only, fallback) ○ Direct2D (windows, default) ○ OpenGL ○ Vulkan ○ platform-specific APIs if none of the above is available
REQ-3	3D audio	platform	○ play audio based on distance to listener ○ support audio filters
REQ-4	built-in debugging tools	-	○ show screen with live-diagnostics for all engine components ○ show system information (hardware, host system, ...) ○ allow adjusting what is shown on the debug screen ○ debug screen is accessible with special key combination (unless debug screen is completely disabled) ○ logging to logfile and in-window console ○ allow transferring logs to other targets (**`QWidgets`**, for example) ○ notify user of fatal errors ○ track allocator states, dangling pointers, and memory fragmentation
REQ-5	configuration	-	○ allow configuring engine settings via **`.json`** or **`.xml`** documents
REQ-6	plug'n'play	filesystem	○ pass game root directory as input ○ engine searches for init files and starts the game ○ specify via command-line or (optionally) via in-window GUI ○ notify user of loading errors

REQ-7	scripting	-	○ scripting of all game entities, GUIs, and other content with integrated scripting language ○ optimized to be easy to use, fast, with syntax ○ support asynchronous operation
REQ-8	media playback	filesystem	○ playback movies, music,
REQ-9	resource streaming	filesystem	○ stream resources into memory as needed instead of loading all needed resources into memory
REQ-10	cutscenes	-	○ support cutscenes that are scripted and derive from the main map (e.g., a fight screen, ...) ○ support interactive cutscenes
REQ-11	animations	-	○ sprite sheet animations with custom framerate ○ property animation for sprites ○ easing curves
REQ-12	save-game	filesystem	○ unified save-game interface ○ support arbitrary size and complexity
REQ-13	graphical effects & shaders	rendering tech used & performance	○ support custom effects via shaders where possible (blur, blending, filters, ...) ○ support HDR where possible ○ support alpha-blending and layered composition methods ○ implement particle system on platforms feasible performance-wise
REQ-14	localization	-	○ dynamically load dialogue translation according to settings ○ also translate built-in menus and dialogues

5. Non-functional Requirements

This section deals with requirements that specify **HOW** the system is supposed to behave as opposed to **WHAT** the system is supposed to do.

5.1. ISO/IEC 25010:2011 Software Quality Attributes

Below is a list of the **ISO/IEC 25010:2011** software quality requirements alongside an integer value, the level of importance, in the closed interval **[1, 5]** where **5** denotes very high importance, and **1** denotes low importance relative to level **5**.

ATTRIBUTE	DESCRIPTION	PRIORITY
FUNCTIONAL SUITABILITY	This characteristic represents the degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions.	**3**
PERFORMANCE EFFICIENCY	This characteristic represents the performance relative to the number of resources used under stated conditions.	**4**
COMPATIBILITY	Degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions while sharing the same hardware or software environment.	**5**
USABILITY	Degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.	**3**
RELIABILITY	Degree to which a system, product or component performs specified functions under specified conditions for a specified period.	**2**
SECURITY	Degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization.	**1**
MAINTAINABILITY	This characteristic represents the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements.	**3**
PORTABILITY	Degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another.	**5**

6. Other Requirements

6.1. Debug Tools

For purposes of monitoring component metrics and behavior, finding errors, and evaluating performance characteristics, Noriko must implement in-engine real-time debugging tools.

6.1.1. Main Debug Screen

The main debugging tool is the **`Debug Screen`**. It can be activated when the engine is running by pressing **`F1`** or **`CTRL+D`**. It shows real-time information on all major components, such as memory consumption, allocator states, renderer states as well as system information (hardware and software), and more. To hide the debug screen, press **`F1`** or **`CTRL+D`** again.

By default, the debug screen needs to show the following information:

I. engine information
   a. product name
   b. version
   c. build date
   d. build flags
   e. compiler used
   f. architecture

II. system information
   a. host platform (**`name`**, **`version`**, **`build number`**, **`architecture`**)
   b. CPU (**`name`**, **`vendor`**, **`clock`**, **`number of cores`**)
   c. RAM (**`size`**, **`clock`**, **`usage in MB and percent`**)
   d. GPU (**`name`**, **`vendor`**, **`clock`**, **`size of memory`**, **`interface`**)
   e. Screen (**`model`**, **`vendor`**, **`refresh rate`**, **`HDR support`**)

III. software metrics
   a. **`FPS`** and **`frame-time`**
   b. viewport size (**`pixels`**, **`tiles`**)
   c. software uptime (**`HH:MM:SS`**)
   d. memory usage of engine (in **`MB`** and **`percent`**)
   e. CPU usage (**`per core used`**)

f. allocator states (**`per allocator`**)
g. renderer state (**`API`**, **`version`**, **`number of GPU resources allocated`**, **`V-Sync state`**)

IV. world metrics
a. current level (**`name`**, **`size`**, **`size on disk`**, **`number of resources loaded`**)
b. current position on level **`(x, y)`**, relative to **`(0, 0)`** [upper-left corner of level]
c. current chunk (**`ID`**, **`position`** (relative to **`(0, 0)`** [upper-left corner of level]))
d. current world time
e. character heading
f. updates/s for chunks
g. what (and how many) chunks are loaded (**`position of chunk in world as ID`**)
h. **`ID`** of current tile (at player position)
i. current **`game file path`**

It must be possible to hide information from the debug screen. This can be done in the engine configuration.

6.1.2. Debug Console

The engine should also feature a debug console which can be shown with the **~** 'tilde' key and hidden with the same button. The console should support the following features:

- customizable colors on character level
- multiline messages
- variable position on screen; height in lines
- scrollable
- customizable font properties
- support for commands

Console settings can be altered through the engine configuration feature.

6.1.3. Logs

All input and output to the console should be logged to a disk file in a special directory. This directory can be accessed via the file explorer if the host application supports this.

The logfile should support the following features:

- one logfile per session
- logfile name should be **`noriko_MMDDYY_HHMMSS.log`**
- can be cleared manually or cleared once every week automatically

Like the other debug tools, these settings can be altered through the engine configuration component. The above settings are the default values.

6.1.4. Crash Dumps

When a fatal error is raised, or the engine process aborts itself, the engine must, before exiting, compile a state dump, dumping call stack, debug screen contents, and process metrics. This crash dump is then written to the disk in a special directory. Crash dumps are not cleared automatically and can only be cleared manually.

7. Optional Requirements

The table below lists requirements that are considered optional, that is, they can be implemented to enhance or introduce existing product functionalities, but are not required for optimal operation. The table layout is similar to the one found in section *4. System Features*.

OPT-REQ-N	BRIEF	DEP	DESCRIPTION
OPT-REQ-1	graphical effects	platform performance	○ some graphical effects are not feasibly supportable on some target platforms due to hardware or software limitations ○ effects that prove to be unfeasible to implement are become optional

8. Future Requirements and Plans

Concrete future requirements are currently not known. A considerable area of improvement would be to implement support for additional rendering as well as audio rendering technologies to improve cross-platform compatibility.

Possible additional target platforms:

- Nintendo® DS®/3DS®
- Google® Android®
- Apple® iOS®
- Apple® iPadOS®
- Sony® PlayStation®
- Microsoft® Xbox®

Appendix A: Original User Story

Games have risen tremendously in popularity and relevance in the past decades. The abundance of genres and titles offers entertainment across demographics and nations. Not only games themselves have risen in popularity, interest in game design and implementation has also seen a big bump.

Noriko is a cross-platform 2-D RPG game engine, focused on capturing the charm of old pixel-art tile-based RPGs that were once popular on handheld platforms from the late 1980s until late 2010s. The engine must support the developer with all the tools they need in order to create striking worlds, addressing shortcomings of popular similar solutions like **`RPGMaker`**.

Feature overview:

- create 2-D tile-based worlds
- support animations
- support cutscenes
- 3-D audio
- visual effects
- cross-platform, targeting not only desktop but also handheld console and device market

The software is created in the hope that it will be useful for people across the globe in taking their first steps in amateur- as well as commercial game development.

Appendix B: Analysis Models

This section contains the main analysis models that have been generated as a result of the analysis phase. They give an overview of Noriko's software architecture as a whole. For detailed documentation and description of each element, refer to the **`developer documentation`**.

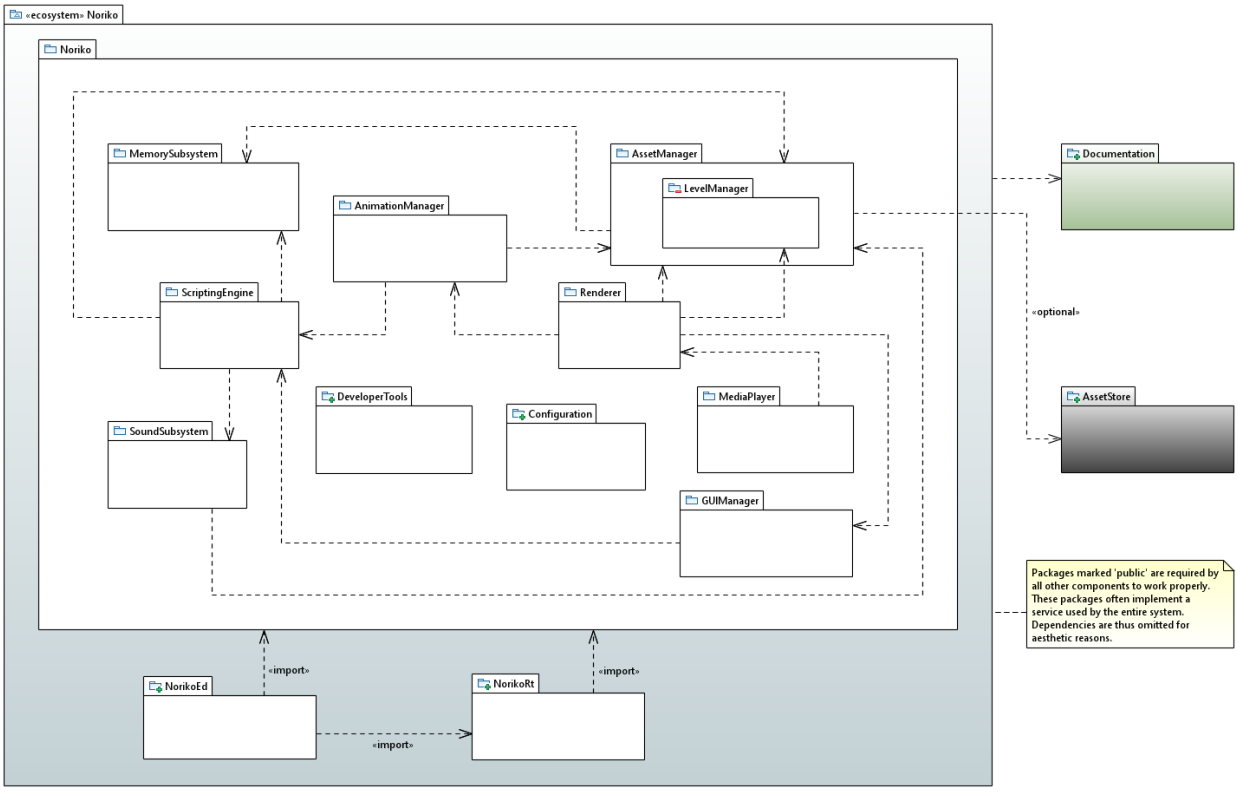

Figure 3: package diagram illustrating Noriko's top-level architecture

The above package diagram illustrates major dependencies, this time with functionality structured in packages rather than components. Isolated packages are assumed to be used by the entire engine. Grey-shaded packages represent optional functionality. Green-shaded packages signify that the functionality encapsulated by the package is not optional but lies outside the main ecosystem.

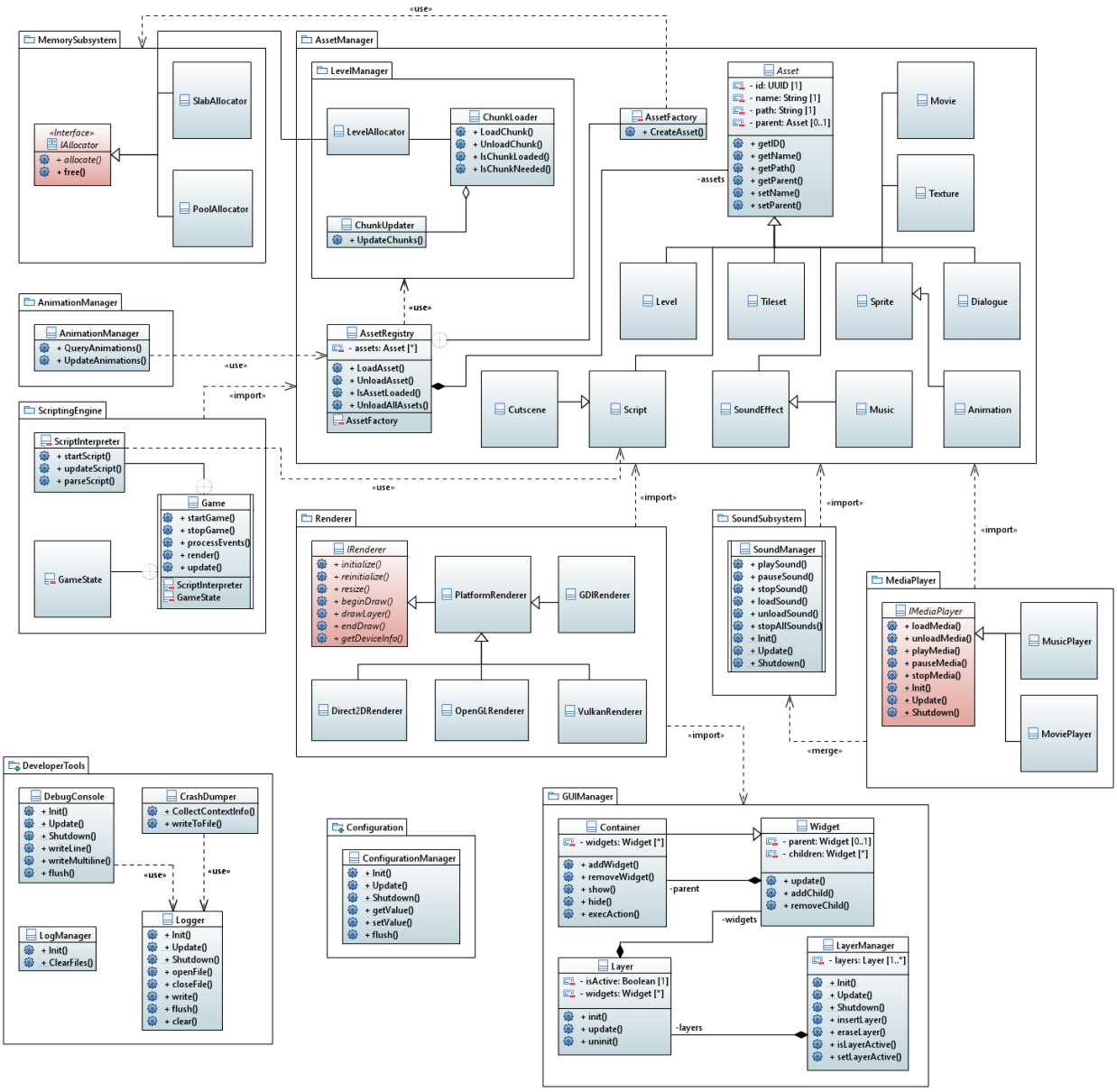

Figure 4: OOA class diagram (CD) illustrating Noriko's engine component's internal structure

The above class diagram is deliberately opaque as it only serves as a slightly more detailed variant of the provided package diagram (**PKGD**) on the previous page. It shows course-grained communication and import relationships of Noriko Engine's major sub-components. This design is still subject to change and may substantially differ in the Noriko's initial release version.

Appendix C: TBD

This section contains items and subjects that are still due for discussion and evaluation. Upon release of Version **`1.0.0`**, this section must be empty save for this notice.