Lappeenranta University of Technology
School of Business and Management
Degree Programme in Computer Science

Bachelor's thesis

Jonni Hanski

STRUCTURE PROPOSAL FOR A VIDEO GAME MODIFICATION MANAGEMENT UTILITY

Examiners: Ossi Taipale
Timo Hynninen

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto

School of Business and Management

Tietotekniikan koulutusohjelma

Jonni Hanski

Ehdotus videopelin modifikaationhallintatyökalun rakenteeksi

Kandidaatintyö

2018

28 sivua, 8 kuvaa

Työn tarkastajat: Ossi Taipale

Timo Hynninen

Hakusanat: modification management utility, design proposal

Keywords: modification management utility, design proposal

Tämän kandidaatintyön tavoitteena oli muodostaa ehdotus käyttäjien videopeleihin tuottaman sisällön hallintaan suunnatun avoimen lähdekoodin ohjelman rakenteeksi. Tavoitteena oli lyhyesti esitellä käyttäjien luoma sisältö konseptina, perustella hallintatyökalun hyödyllisyys yleisellä tasolla ja muodostaa sekä perustella ehdotus kyseisen kaltaisen työkalun korkean tason rakenteeksi. Rakenteen tavoitteena oli olla yleisen tason rakenne, joka ei turhaan sido käytännön toteutusta. Kandidaatintyön toteutus perustui olemassa oleviin julkaisuihin, jotka käsittelivät muun muassa käyttäjien tuottamaa sisältöä ja videopelien modeja, ohjelmiston mielekästä rakennetta yleisesti sekä ohjelmiston rakenteen vaikutusta avoimen lähdekoodin ohjelmistoprojektiin. Työn tuloksena pystyttiin muodostamaan ehdotus kyseisen kaltaisen työkalun rakenteeksi. Uuden rakenteen tavoitteena on taustakirjallisuudessa esitettyjen näkemysten viitoittamana edesauttaa kyseisen kaltaisen työkalun kehitystyötä.

ABSTRACT

Lappeenranta University of Technology

School of Business and Management

Degree Programme in Computer Science

Jonni Hanski

Structure Proposal for a Video Game Modification Management Utility

Bachelor's thesis

2018

28 pages, 8 figures

Examiners: Ossi Taipale

Timo Hynninen

Keywords: modification management utility, design proposal

The aim of this bachelor's thesis was to formulate a proposal for the structure of an open-source utility used in managing content created by users for video games. The bachelor's thesis sought to briefly introduce the concept of mods as user-generated content, to justify the need for an automated utility to manage such content and to finally formulate and justify a proposal for a high-level structure for such a management utility. The idea was to formulate the structure in a way that does not impose restrictions on the actual implementation of the structure in an unnecessary fashion. The work on the thesis was based on existing publications about, for example, user-generated content and/or video game mods, reasonable software design and structure in general, as well as the effect of the structure of a piece of software on the open-source software project surrounding it. The result of the work was a proposal for a high-level design for the type of utility in question. The formulated design seeks to facilitate reasonable development of the utility, guided by the views presented in the publications used as background material.

PREFACE

Working on this thesis has been an interesting learning experience when it comes to writing things down in a more official capacity than usual. Overall, I could perhaps argue that I have learned quite a few things with regards to searching for sources, referencing them, designing the layout and structure of my writings and trying to make things somewhat understandable to others. However, I have also realised that there is still an awful lot to learn and that my journey has only just begun. Wherever I will continue from here, the only thing I can know for certain is that it will take quite a lot of learning to get there!

As for this bachelor's thesis specifically, I would very much like to thank my supervisors Timo Hynninen and Ossi Taipale for helping me throughout my work on the thesis, by providing excellent ideas, commentary, examples and overall guidance to help me get things done, even though I may sometimes have had some minor issues trying to follow all the recommendations and tips, for example with regards to schedule and other little things like that. Also thank you for your seemingly endless patience! In addition to my excellent supervisors, I would also like to thank Ari Happonen for arranging the course to help everyone, including me, write their bachelor's thesis and for providing overall guidance and other tips for the creative process. Also thank you to everyone who has ever taught me, including all university lecturers, tutors and all my teachers from primary school through to university for helping me get this far. Additionally, an absolutely overwhelming thank you to my family who have supported me, who have been very encouraging and who are definitely the best family one could ever have wished for! Also thank you to my brother for catching some (that is, a ridiculous heap of) obvious typos within my thesis that I had completely missed myself. Hopefully there are less typos left now than there were before. As was mentioned, I still have a lot to learn.

TABLE OF CONTENTS

ABBREVIATIONS

DLC	downloadable content
API	application programming interface
mod	modification (abbreviated, a form of user-generated content)

1 INTRODUCTION

For the purposes of establishing a preliminary understanding of mods as a concept, as well as the concept of using an automated utility for managing them, it could be purposeful to refer to existing publications that could help illustrate the concepts better. Any concepts will only be described to the extent required for the purposes of the thesis, however. The motivations and structure of the thesis will also be introduced briefly.

1.1. Video Game Modification

Scacchi (2011) uses the term mod to refer to user modifications to video games and the term modding to describe “the practice and process of developing game mods”, that he further describes as “an approach to end-user game software engineering”. Scacchi further elaborates on the concept of mods by presenting them as a potential form of open-source software used to extend closed-source or open-source software, that is, user-created open-source extensions to an open-source or closed-source video game. El-Nasr and Smith (2006) have additionally explored “the use of modifying, or modding, existing games as a means to learn computer science, mathematics, physics, and aesthetic principles.” According to El-Nasr and Smith, such activities have been enabled by “a recent increase in the number of game environments or engines that allow users to customize their gaming experiences by building and expanding game behavior.”

For the purposes of this thesis, the term mod, also potentially referred to within this thesis as video game modification, video game mod or anything similar, shall be used to refer to modifications to a video game, that are carried out after the release of the video game, that are not part of the official development effort of the video game, that can affect the appearance or behaviour of the video game or otherwise change the structure of it, that may be created by any entity, including, but not limited to: individual people, groups of people or automated software. For example, changes to any functional or artistic content, story, items, gameplay, setting or any other content within a video game or within the game engine itself, that are not performed for any illegal purposes or in any potentially malicious fashion whatsoever, shall be referred to as mods. For example, improved graphics assets, enhancements to storyline or dialogue, addition of new items or removal or editing of existing items such as armour, weapons or locations within the game world, could be included in a mod. Following the views of Lee et al. (2015), new content to a video game may either be downloadable content (DLC) by official developers or community-generated mods, of which the latter reflects the type of new unofficial content referred to within this thesis.

1.2. Video Game Modification Management Utility

The term video game modification management utility, potentially referred to within this thesis also as mod management utility, mod manager, utility or any other equivalent term, shall be used to describe a piece of software crafted specifically for the purposes of managing data related to video game mods on a user's computer. Some existing example modification management utilities include, but are not limited to: Wrye Bash, Mod Organizer and Vortex. Being open-source software, the source code and documentation for each of the listed examples is freely available on their respective GitHub repositories (sources listing, items 12, 13, 14). Based on the documentation and source code of the example mod management utilities, one may assume that mod management may include, but may not be limited to, for example:

- Addition and/or removal of data to and/or from a video game within the context of adding or removing changes, content or other data as a part of mod management activities such as installing or uninstalling (applying or de-applying) mods or other edits.
- Automated (either entirely or partially) acquisition of mod data from various sources, such as online data hosting providers, to the user's local computer, as well as optionally including the ability to offer either partially or wholly automated or guided installation processes for various mods in the fashion of traditional software installation wizards.
- Support for advanced tools or extensions for carrying out more complex tasks related to various data associated with a game or mods, for example for the purposes of editing a mod, managing game saves or for compatibility troubleshooting.
- Preferably support for multiple different video games and operating systems, making it possible to use a single utility that can be extended as deemed necessary to support new games and operating systems, instead of developing a completely new separate tool for each video game or operating system, as there may end up being overlapping common fundamental functionality in each separate utility and developing the same functionality in a slightly different fashion multiple times may not be an efficient way to use voluntary development resources.
- Optionally including support for multiple ways of adding data to a video game, for example by directly embedding the data within the game, or by linking or otherwise redirecting data. Examples may include directly copying files on disk, using symbolic or hard links, or using some sort of virtual filesystem overlay or hooks to redirect process filesystem access.

- Support for extending the functionality of the utility in either game-specific or global fashion, for example by creating new extensions to carry out more complex or more specific tasks for various purposes.

Based on the activities a mod management utility may be required to perform, one may seek to build a preliminary list of functionality that should be present within such a utility. Some example functionality with regards to implementation considerations for the purposes of this thesis may include, but may not be limited to:

- Implementation of support for various games.
- Implementation of support for various mod data formats, as well as mod data installation and uninstallation methods.
- Implementation of support for various online data hosting providers.
- Implementation of support for various automated installation wizard declaration formats, for example for the purposes of supporting existing automated mod installation wizard formats as well as any future formats.

The various functionalities provided by a mod management utility may be considered more of an added layer of convenience and abstraction, for example one might argue that downloading data manually (of sorts) from an online hosting provider and then adding that data to a video game may sometimes be doable without an added piece of software. One may also assume, however, that an added element of convenience could, in some cases, help reduce the time or expertise required for adding mods to a video game and could therefore allow for the direction of more time and effort of an individual seeking to manage mods for their game to various other activities that may involve less routine work that can be automated. Representing mods to user as abstract, complete and uniform packages that can be manipulated, as opposed to having the user manage various individual data items, may offer a different mod management experience. Reducing the amount of expertise required, for example if a video game uses a command line utility for unpacking, altering and then re-packing data, by automating the process and adding a layer of abstraction between the technicalities and the end user may perhaps help make the use of mods more accessible to individuals who prefer operating at a higher level of abstraction.

1.3. Motivation Behind the Thesis

Motivation behind the thesis mostly revolves around the current perceived state of an example modification management utility, namely Wrye Bash, a free, open-source modification management

utility for a handful of video games. The source code of the utility is located at GitHub for anyone to view or contribute to (sources listing, item 12). According to the Wrye Bash repository at GitHub, it is “a mod management utility for [The Elder Scrolls IV:] Oblivion and [The Elder Scrolls V:] Skyrim with a rich set of features.” Potential developers would appear to be encouraged to participate in developing the tool, and guidelines seem to have been created for contributions, with the first handful of contributions from each new developer seemingly being checked first by a current developer for compliance to help get everyone on track (*Contributing to Wrye Bash*). The utility itself is also of a certain age, with the version history listing version 0.01 as the first release on 13 April 2006. The latest considerably revised version listed at the time of writing would appear to be version 306 with major changes including code refactoring and performance improvements from November 2015 (*306 Code refactoring, performance*, Utumno 2015). Additionally, the version history (*Wrye Bash Version History*) lists a plenitude of developers who have contributed to the utility over the years.

Based on the available source code of the utility, it would appear to carry out mod management activities by copying and removing files or parts of files belonging to mods on a user’s hard drive. All of the supported games would appear to use similar file path structures for mod files, something that is indicated by the current implementations of game-specific modules seemingly not providing their own directory paths to where mod files are to be located within an installation location of a game. Wrye Bash also appears to offer functionality for managing game configuration files, resolving conflicts between mods, managing game saves and for using an automated installation wizard for a mod to guide the user throughout the installation process by presenting sets of choices to the user as defined in an installation wizard description file should such a file be present and valid. At the time of writing, according to the documentation and the source code at the GitHub repository (sources listing, item 12), Wrye Bash seems to have been implemented using Python 2 with wxWidgets for the graphical user interface through the wxPython libraries. Other libraries seem to be used to extend Python 2 to provide some additional required functionality, such as pywin32 to provide access to Microsoft Windows API functionality.

The current developers, at the time of writing, would appear to be working on cleaning up and restructuring the source code of the utility, and there are projects listed on the GitHub repository aimed at improving the structure of the utility (*Wrye Bash GitHub repository, Issue #200*). One such project seems to be an attempt to collect all references to the graphical user interface framework to a single location for the purposes of facilitating a potential easier transitioning to a new version of the framework at some point in future without too much hassle (*Wrye Bash GitHub repository, Issues #63 and #190*). The contribution guidelines instruct contributors to write Python 3 compatible code

whenever possible to prepare for a potential upgrade of the language version from Python 2 to Python 3 (*Wrye Bash GitHub repository, Contributing to Wrye Bash*). Several different implementations in separate locations for a single functionality, as well as some performance issues seem to have been identified and resolved recently in the source code (*306 Code refactoring, performance*, Utumno 2015). One might additionally argue that, from a subjective code readability perspective, the naming of a number of files/modules, functions and variables in the current codebase may not necessarily reflect their purpose, with module names, for example, including *bash.py*, *bosh.py*, *balt.py* and similar, although the intended contents and purpose of all such modules also seem to have been provided (*Wrye Bash GitHub repository, Deciphering Source Filenames*). Cross-platform support would appear to be somewhat limited (*Wrye Bash GitHub repository, Running Wrye Bash on WINE (Arch Linux)*), the user interface framework seems to be an older version—something that is reflected by the plans to upgrade its version—and the interpreted Python language used is a legacy version according to the language documentation (*Should I use Python 2 or Python 3 for my development activity?*, Python Wiki).

While the developers of Wrye Bash evidently seem to be willing to invest time and effort in further developing the utility, based on their activity over at the GitHub repository, the design and codebase of the utility itself may impose potential obstacles that the developers might need to overcome to successfully carry the utility onwards, something that appears to be demonstrated by their work on refactoring and restructuring the source code to be able to reach their development objectives, such as upgrading the graphical user interface library and Python language version. This potential technical debt, that would appear to be affecting further development, raises the question of whether it would be possible to rethink the overall design of the utility to mitigate some potential obstacles related to extending the utility, further developing it or upgrading the components used now and in the future, and whether it would then be possible to simply rewrite some or all of the utility in a way that adheres to any reasonable coding guidelines, uses modern frameworks, languages and tools to remove any technical debt and follows a new, improved design from the very beginning.

While neither the thesis itself nor the author of it are in any capacity connected to the development team of Wrye Bash, the concept of drafting an improved design that would help overcome and prevent some of the issues experienced by Wrye Bash seemed like an interesting idea worth exploring. Drafting a speculative proposal, as well as potentially developing a smaller scale prototype based on it, may or may not also help one gain a better understanding of the challenges faced by a modification management utility project such as Wrye Bash, in addition to potentially solving some of the issues to some extent.

1.4. Objectives and Limitations of the Thesis

The objective of the thesis is to formulate a design proposal for a video game modification management utility, taking into account some general software design considerations and considerations when building and developing software as a voluntary open-source software project. The design proposal is to offer a high level structure and/or guidelines for the development of utility without imposing any unnecessary restrictions on the practical implementation of such a proposed design. No complete design diagrams are to be drafted, and illustrations shall only be provided for illustrative purposes and may not necessarily be accurate enough to facilitate practical implementation without additional design effort. This is to keep the thesis compact enough and to avoid locking the utility in any specific implementation methods through design.

The resulting potential overall design proposal shall remain a speculative proposal that is merely an idea and is in no way requested by, affiliated with, connected to or endorsed by any entity whatsoever, although any ideas or parts of any ideas presented herein can freely be adopted by the developers of any video game modification management utility, without providing any credit, should the ideas prove useful or otherwise interesting.

The research questions, the answers to which shall potentially be approached using a variety of obscure speculations and other suspicious and/or dubious methods that may or may not actually make any sense, could be described as follows:

- What type of design could be reasonable for software in general, with regards to the feasibility of maintenance and development?
- What type of design could be reasonable for software developed as a voluntary open-source software project, for example with regards to facilitating simultaneous development by various, largely independent volunteering developers?
- What type of design would be purposeful for a video game modification management utility specifically, with regards to the feasibility of maintenance and other development of the utility, with the utility being developed as a voluntary open-source software project?

Any exact or exhaustive definitions of video game modification, user-generated content or modification management utility shall not be provided or collected. Instead, only definitions or descriptions needed to establish a general understanding required for the purposes of the thesis will be provided or referenced. The creation of video game modifications or similar content is likewise beyond the scope of the thesis, as is the practical implementation of the management utility and the management of any implementation project, as well.

Sources used throughout the thesis are ones found through reasonable research effort and may not necessarily represent all relevant sources available. It is not at all impossible for sources that would have been relevant to be absent, however the current sources used should be enough for the purposes of the thesis.

1.5. Structure of the Thesis

Chapter one features a brief introduction to the thesis itself, as well as to the concept of video game modifications as a form of user-generated content. The concept of video game modification management utility is also introduced briefly, as are the motivations, objectives, limitations and the structure of the thesis. Chapter two seeks to establish a general understanding of reasonable software design in a way that facilitates the development of the software within the context of an open-source software project as a voluntary effort by individual developers. Chapter three introduces the proposed design for a modification management utility based on previous conclusions and notes, also seeking to account for the nature of a modification management utility and the development of one. Chapters four and five include conclusions and thoughts on potential further development of the design, as well as the sources used in writing this thesis.

2 SOFTWARE DESIGN AND OPEN-SOURCE SOFTWARE PROJECTS

For the purposes of drafting a reasonable design proposal for a modification management utility, it could be beneficial to take a brief look at existing publications related to software design and open-source software projects to establish a preliminary understanding of the nature of a potentially reasonable design for a piece of software developed as an open-source project.

2.1. General Software Design

As presented by Yourdon and Constantine (1979, pp.16-26), dividing a problem in manageably small, separately solvable parts can help minimise the cost of implementing a computer system. Similarly, keeping the parts of the system easily related to the application, manageably small and separately correctable can help minimise the cost of maintenance of the system. Additionally, the cost of modification of a system can be minimised by keeping its parts easily related to the problem and modifiable separately. To quote Yourdon and Constantine (1979): “Implementation, maintenance, and modification generally will be minimized when each piece of the system corresponds to exactly one small, well-defined piece of the problem, and each relationship between a system’s pieces corresponds only to a relationship between pieces of the problem.” Additionally, structured design should help lead to minimum costs when highly interrelated parts of a system reside in the same piece of the system and unrelated parts reside in unrelated pieces of the system and have nothing to do with one another. Yourdon and Constantine note that simple structure for a computer program needs to be designed that way from the beginning, and any modularisation, that is, breaking an existing, already implemented computer system into smaller pieces, does not necessarily help make the system easier to manage or less complex. Based on the views by Yourdon and Constantine, one may conclude that by breaking a software down into multiple independent components from the very beginning may help reduce the amount of resources needed for implementing, maintaining and modifying a piece of software.

2.2. Open-Source Software Project and Software Design

Based on the views presented by Peng, Geng and Lin (2012), as referenced by Haapaviita, et al. (2013), from the perspective of an open-source software project, a modular design where software is broken down into multiple independent components could have the potential to allow for each developer to work on the field they themselves know best, by keeping their specialised functionality to their own components. Parallel development could also be facilitated, to improve development speed and remove or decrease delays caused by overlapping changes. Speculating further, one may

also assume that isolating groups of functionality could also help better keep the source code organised in a semantic fashion on a file level and also help avoid accidentally editing anything unrelated when developing one specific set of functionality, in case someone unfamiliar with the utility software seeks to contribute to it. Van der Hoek and Lopez (2011) argue, referencing Lee (1997), that “much of a developer’s work involves interpreting code that is already in existence.” To that end, one would assume that a modular design for a software system could potentially help developers better reason about the system and detect the areas of interest to the current development effort.

In addition to modular design, an open-source software system could perhaps benefit from a straightforward way of extending the software with the use of dynamically loaded plug-ins through the use of a documented extension interface. Following the views of Mayer et al. (2003), when using a plug-in architecture, “nothing referring to a specific plug-in is hard coded into the application’s source code”, which one could assume to be a factor that may facilitate reasonable removal, addition or replacement of functionality within a software system without having to edit the rest of the system. According to Ye and Kishida (2003), as referenced by Haapaviita et al. (2013), a plug-in architecture of sorts could help new developers join a project that they have limited knowledge about. One may assume that, for example, a straightforward, documented way to develop plug-ins to interact with the main application and extend it could help lower the barrier of entry for someone seeking to contribute by removing the need to learn all the utility’s internals before being able to contribute.

2.3. Implications for Modification Management Utility Design

Based on the views of Yourdon and Constantine (1979) on modular software design, as well as the views of Peng, Geng and Lin (2012), as referenced by Haapaviita, et al. (2013), one may conclude that a modular design could be reasonable for a modification management utility as an open-source voluntary software project with a limited amount of resources at its disposal, with potentially numerous contributors and functionality that should adapt to numerous slightly different use cases. One may assume that a modular design could also allow independent developers to concentrate on the aspects of the software they themselves have the skills to contribute to, without having to exert excessive communication and conflict resolution with other developers when editing unrelated parts of the software. Additionally, following the views of Mayer et al. (2003), as well as Ye and Kishida (2003), as referenced by Haapaviita et al. (2013), a plug-in architecture could help reduce the barrier for external developers to contribute new functionality or features by allowing developers to merely plug a new functionality module into a well-documented interface within the utility.

3 VIDEO GAME MODIFICATION MANAGEMENT UTILITY DESIGN PROPOSAL

Having established a general understanding of a reasonable design for software being developed as an open-source software project, a design proposal for a video game modification management utility may now be drafted, on a general level, without unnecessarily restricting practical implementation, taking into account previously noted software design guidelines and views.

3.1. Overall Design Considerations

Based on the source code and documentation of existing modification management utilities, such as the ones listed earlier for example purposes, one might conclude that the functionality required of a modification management utility may change as a result of, for example, adding, removing or changing any of the following: the game to manage mods for, a feature of the utility itself, a supported format for mods, files or other data, a process related to mod management or any external service or software integration. Any changes in functionality might in turn translate to changes in the code contained within the utility. To facilitate these changes, a modular design could be beneficial, as was concluded within chapter two. The modular design itself could help developers better manage logical groups of functionality, when each set of specialised functionality could be isolated in its own module. Such specialised functionality could include, for example, the main graphical user interface, game-specific data management, support for various mod and other data formats, as well as other such logical groups that one would develop independently to carry out a specific task without excessive interdependency with other modules. Additionally, while a modular design in and of itself may be purposeful for the utility as an open-source voluntary software project, the design could also benefit from being such that would allow for the replacement, removal and addition of modules/functionality without adding additional overhead to the development, maintenance, deployment or usage of the utility itself. Having each set of specialised functionality isolated in its own module could make it more straightforward to swap such modules in and out depending on current or future needs. For the purposes of formulating a design proposal, some example modules have been presented within figure 1 for illustrative purposes only.

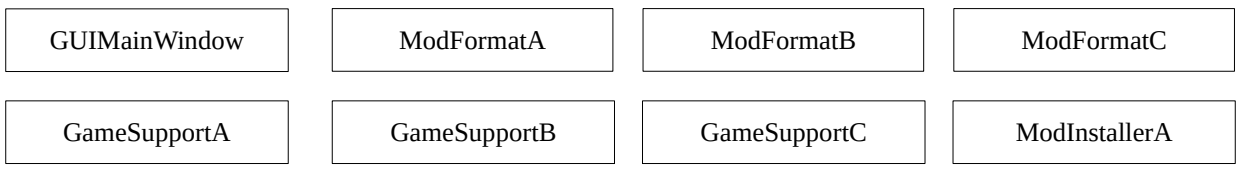

Figure 1: Example modules for illustrative purposes

3.2. Modularity and the Monolith

While modular design itself would allow the developers to isolate functionality in various specialised modules that could be developed independently and that could communicate with one another through commonly agreed interfaces, the modular design of the utility should also be one that allows for effortless adjustment of functionality as needed to avoid adding overhead to the development or deployment of the utility to conserve developer resources and to avoid placing additional strain on individual users of the utility. For example, if different supported games require different sets of functionality, then supporting multiple games might require adjusting functionality on a game-by-game basis. Additionally, the naïve solution of connecting modules with one another through various module-specific interfaces may facilitate the construction of a difficult-to-track web of hard-coded interdependencies between various modules, unless developers take special care to minimise coupling between modules. This minimised coupling should also be something new contributors should pay attention to, for example individuals who seek to contribute a small component such as integration with only a single external tool or service. Without careful consideration, the modular design may not be able to prevent the accidental construction of a monolithic construct comprised of interdependent modules. Figure 2 attempts to illustrate this.

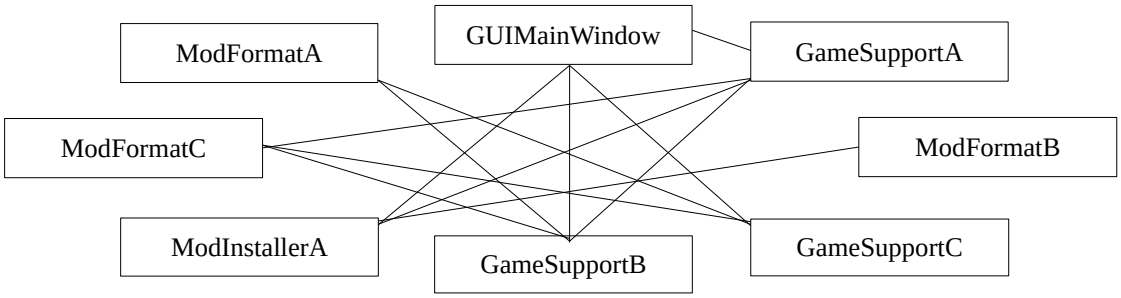

Figure 2: Monolithic construct comprised of modules woven within a web of uncontrolled interdependencies.

Using a heavily interconnected and interdependent set of modules implemented as files that are included and imported into a monolithic mod manager, the process of replacing, adding or removing functionality could involve adjusting module imports, replacing modules and editing various aspects of the utility as it is being prepared for release. Releasing the utility for a number of different games, for example, might involve editing and preparing a separate release for each game, depending on the differences in required functionality. As the amount of supported games, data formats, tools, services and other functionality would increase, the amount of effort required for preparing releases for each game may also increase. Therefore, the addition or removal of functionality, as well as replacement of overlapping alternative functionality could somehow be automated to reduce human effort.

3.3. Plug-In Architecture and Inter-Plug-In Communication

For the purposes of automating the replacement, removal and addition of modules, implementing the modular design through a dynamic plug-in architecture could be considered. Instead of preparing and releasing a single monolithic modification management utility for each different environment, the utility itself could consist of a single core application component and a set of dynamically loaded plug-ins that could be loaded or left unloaded depending on the current use case of the utility. As noted by Singer (2017), plug-ins "can be used for designing an application that can be customized to a specific client". Singer further elaborates on the usefulness of plug-ins in extending a single piece of software for multiple slightly different use-cases by stating how "necessary customizations could be captured by different modules that plug into a core system." Following the views of Ye and Kishida (2003), as referenced by Haapaviita et al. (2013), a plug-in architecture could also be otherwise beneficial for the utility as a whole, in addition to removing any work required to prepare and release separate versions of the utility. The core application component could perhaps also provide means for individual plug-ins to access functionality provided by other plug-ins through a common central interface to reduce the amount of hard-coded interdependencies. Overall, one could assume that implementing a dynamic plug-in architecture may allow for some or all of the following:

- Individual plug-ins themselves could contain overlapping functionality, for example different implementations of a single functionality, as long as no two plug-ins with overlapping functionality would be loaded simultaneously or as long as there would be a system for overriding functionality provided by other plug-ins.
- Specialised functionality could be isolated into a single plug-in for easier management, development, replacement and testing even. The number of overlapping code edits could be controlled by having each developer only develop a single plug-in.
- Developers would need to ensure that the plug-in loading mechanism during runtime would be such that allows the developers of plug-ins reasonable control over the circumstances under which their plug-ins are loaded, for example the currently managed game, other loaded plug-ins or even the order in which the plug-in is loaded relative to other plug-ins.
- Developers seeking to contribute a new plug-in to the utility could study the documentation provided by other plug-ins' developers to determine the functionality available to their plug-in and where it originates in. The functionality could then be used through the central interface.

- No separate releases for different use cases, such as different games, would necessarily be needed, as the core application and all plug-ins could be shipped only once and the plug-ins themselves or the core application would determine whether specific components are relevant to the current deployment or not.

However, the implementation of such a dynamic system would require that the loading mechanism be sophisticated enough that no plug-in is loaded within an inappropriate scenario and that plug-ins are loaded in a controlled, sequential fashion without breaking anything. The common interface, referred to here as the plug-in API component, would need to be clear and concise and should not be too complex to break anything by itself. The API component would be used to expose functionality to other plug-ins, in that it would act as a mediator to connect functionality from various plug-ins with other plug-ins needing such functionality, in addition to managing the loading of plug-ins. Figure 3 attempts to illustrate this dynamically loaded plug-in idea.

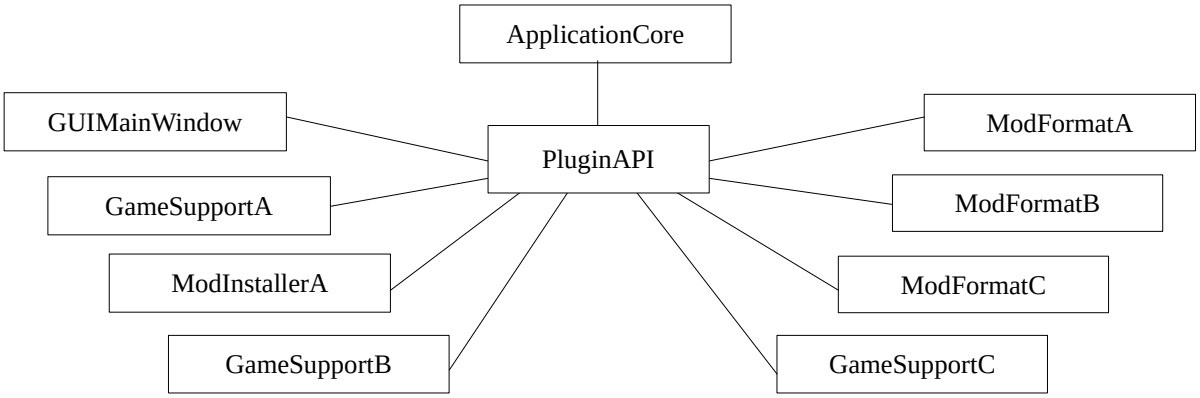

Figure 3: Dynamic plug-in architecture illustration.

One would assume that restricting the interaction of a plug-in within the utility to happen strictly between the plug-in and the plug-in API could help simplify the web of interdependencies within the utility and could perhaps also help developers of new plug-ins better understand the environment for which they are developing their plug-ins when there would only be one point of entry from their plug-in to the rest of the utility. The general idea would also be, as was stated, to limit the amount of direct dependencies of a plug-in, that is, hard-coded references to other plug-ins. With the proposed design, there would be only one direct dependency coded within the source code of each plug-in: the plug-in API. Anything else would be made available to a plug-in by the central API. Figure 4 attempts to illustrate this.

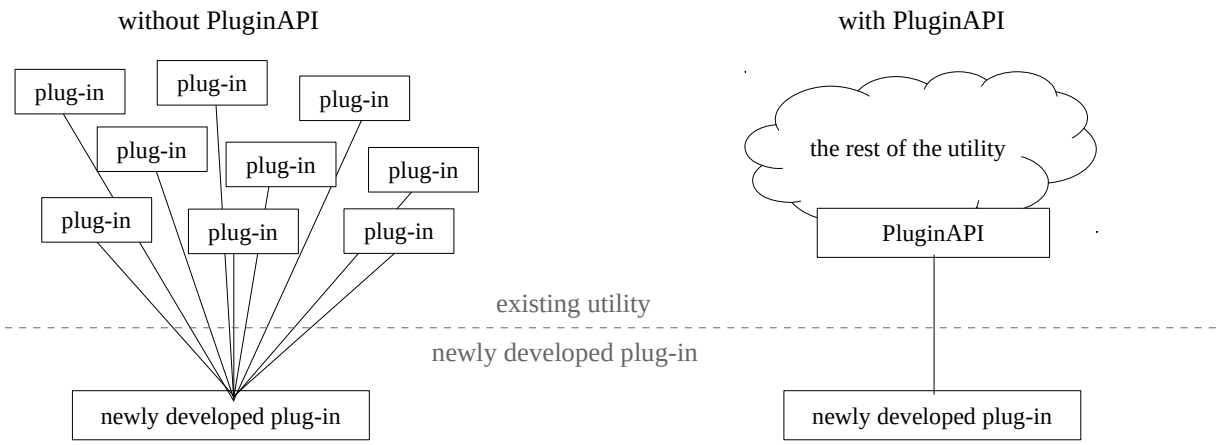

Figure 4: Simplification of plug-in dependencies and interaction.

Communication between various plug-ins would therefore also take place via the plug-in API component, as was mentioned. One would assume that the responsibilities of the plug-in API component could therefore include some or all of the following, however it may not be limited to those specifically, but for example purposes there have been some listed below:

- Dynamically loading plug-ins during runtime, while also providing plug-ins themselves the ability define any requirements for loading them.
- Providing a common, uniform means for plug-ins to request functionality from other plug-ins or the application itself, as well as providing an error handling and/or logging and reporting mechanism in case a piece of required functionality is missing.
- Removing the need for plug-ins to reference one another directly by acting as a central unified mediator between any and all plug-ins within the utility.

Such a central plug-in management system would allow for not only the removal of hard-coded dependencies between plug-ins but also the relocation and reorganisation of functionality between plug-ins, when the functionality would be referenced through the common interface. Various plug-ins could therefore perhaps overwrite functionality provided by others, when anyone needing functionality would query the plug-in manager component for such functionality. Completely new plug-ins could be written to provide new, better implementations for existing functionality and such plug-ins could be integrated into the utility in a seamless fashion without necessitating changes to any other existing plug-ins, as well.

In an effort to facilitate controlled loading of plug-ins during runtime, one would assume that the plug-ins would need to have a mechanism to inform the loading system of any preferences with regards to, for example: the current deployment environment of the utility, load order relative to other plug-ins or requirements related to functionality or other plug-ins being present before being loaded. Should a plug-in require functionality from another one, for example, it would be reasonable to have the plug-in loaded after that other one. The plug-in manager component could therefore perform some form of sequential loading of plug-ins depending on preferences reported by the plug-ins themselves. Additionally, as noted by Singer (2017), a plug-in could be had report the supported API version, as well, should the API be updated or altered. One would therefore conclude that the process of loading plug-ins could feature at least some of the following steps taken by the plug-in management component:

1. Preliminary query to each detected plug-in to establish, for example:
    - The list of plug-ins or other functionality required to be present and loaded before loading the plug-in.
    - The conditions, for example with regards to the name of currently managed video game, that must be met for the plug-in to be loaded.
    - Any other conditions.
2. Sorting and cleaning the list of plug-ins to be loaded depending on the requirements reported by each plug-in, for example removing plug-ins that do not have their loading conditions met and then sorting the remaining plug-ins depending on their requirements.
3. The actual loading of plug-ins in sequential order.

The exact implementation of the loading mechanism and requirements reporting would need to be determined depending on the chosen implementation method. For example, in case a plug-in cannot be loaded even in a preliminary fashion without the environment meeting some conditions, one would assume that the requirements could then be stated within a manifest file of some description accompanying the plug-in. The complexity and robustness of available condition requirements, as well as the specific syntax, would therefore also depend on the method of stating the requirements. Additionally, for the dynamic plug-in loading mechanism to be possible, each plug-in could provide a standard interface for the plug-in manager to load them, so that each one would fit the standard process of loading a plug-in to keep the loading process consistent for all plug-ins. An abstract plug-in class could perhaps be used for such purposes, depending on the implementation method. Figure 5 seeks to illustrate this abstract plug-in base and inheritance concept.

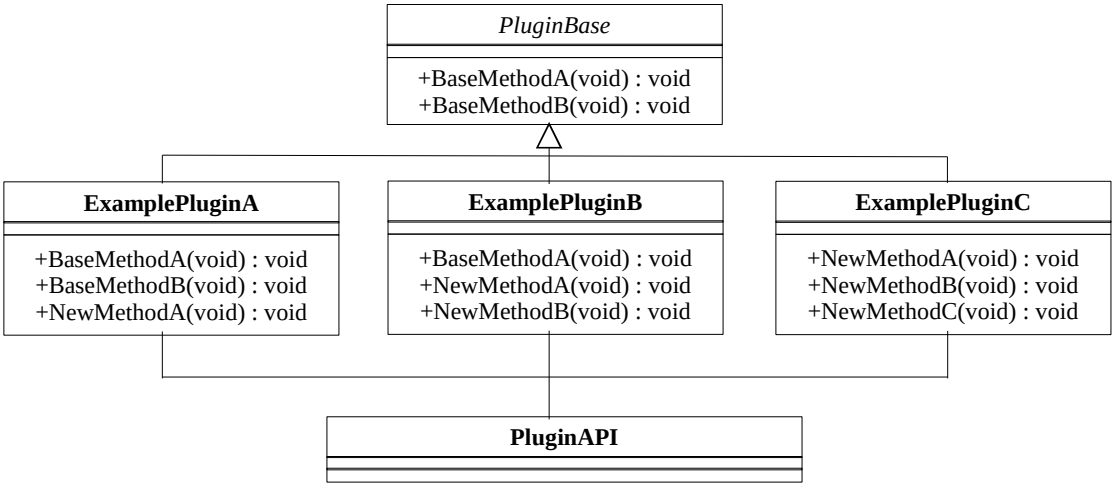

Figure 5: Illustration of the plug-in manager functionality request redirection.

Implementing an abstract base plug-in class might also help develop new plug-ins with less repetitive functionality implementation, when developers could have their own plug-in inherit the abstract one and subsequently be equipped with all functionality required of it, so the developers could concentrate on implementing only the functionality their custom plug-in seeks to offer, by adding new functionality and by overriding some functionality from the abstract class as needed. This would, however, require that the chosen implementation method supports some form of object-oriented programming and inheritance, or that it provides a comparable concept or method for implementing something similar. Without an abstract base plug-in class, however, the system may still be possible to implement, depending on the implementation method.

As for illustrating the process of loading plug-ins, for example in the case of two plug-ins—PluginA and PluginB—the plug-in management component could first perform a query to establish the relevance of each plug-in and the load order of each relative to other plug-ins, after which they would be sorted and loaded sequentially. Figure 6 seeks to illustrate this scenario, where PluginA requires functionality provided by PluginB during its initialisation phase and must therefore be loaded after PluginB. During its loading, PluginB registers this functionality at the PluginAPI, after which PluginA will gain access to it after querying the PluginAPI.

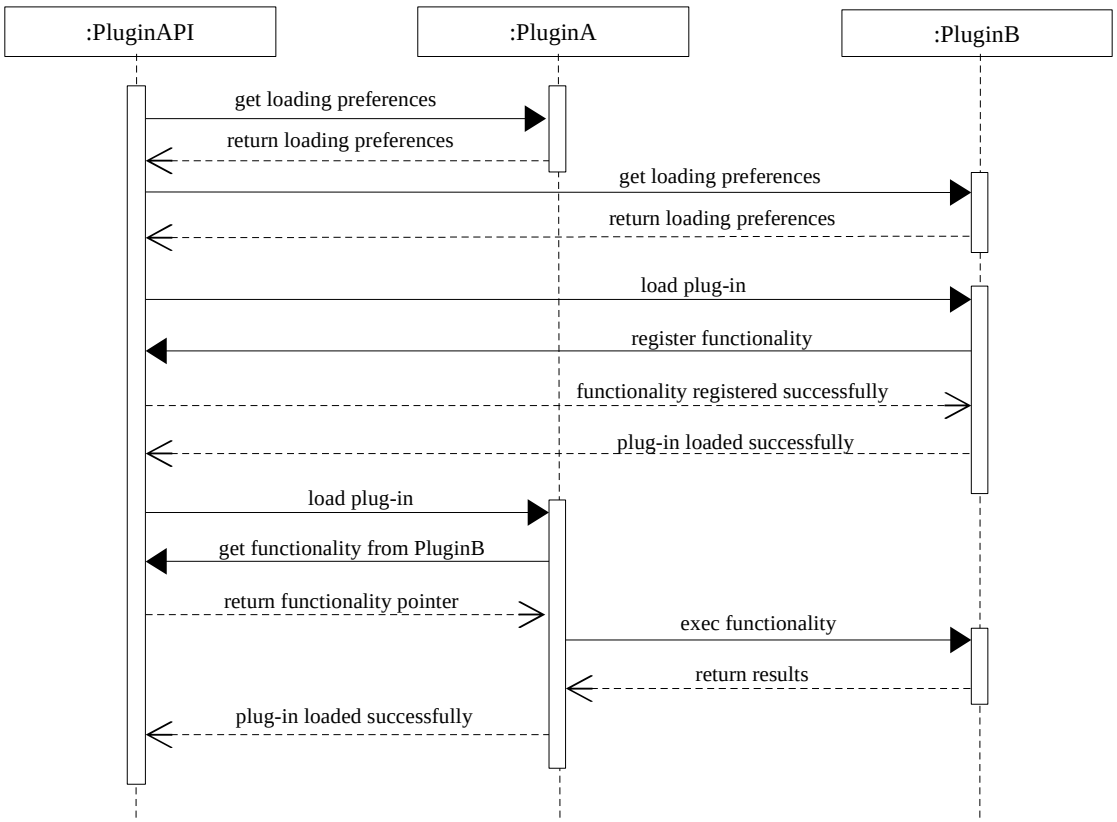

Figure 6: Plug-in loading mechanism illustration, when PluginA requires functionality provided by PluginB and must therefore be loaded after it.

The mechanism for registering and requesting functionality at the plug-in management component would also need to be designed in a way that would allow for plug-ins to register functionality in whichever capacity may be required, depending on the implementation method of the utility. One would assume, however, that the method for registering functionality would be such that would also allow for uniform, consistent access to any registered functionality for all plug-ins. For example, one potential method of registering functionality could include some form of naming system, where each plug-in could register functionality at the plug-in manager component using a string identifier, so that pointers to functionality could be requested using the string identifiers, and the plug-in providing the functionality would not matter, as any plug-in could register functionality using any string identifier.

For illustrative purposes, one may consider the following three plug-ins: PluginA and PluginB that provide functionality and PluginC that uses it, by requesting it from the plug-in management component. The plug-in management component directs (for illustrative purposes) the functionality request to the plug-in that has registered one of its methods as an implementation of the functionality identified by the string. Figure 7 seeks to illustrate this.

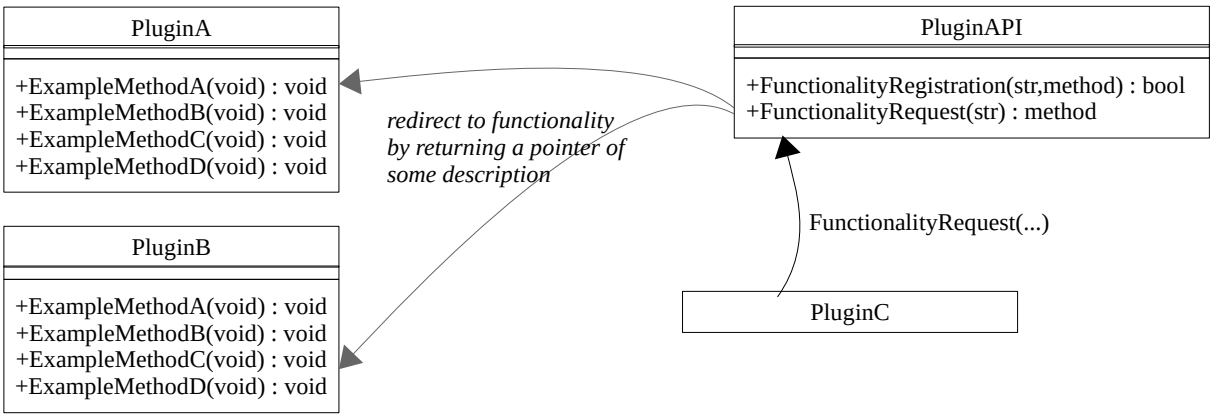

Figure 7: Illustration of the plug-in API functionality request redirection.

One would assume that implementing a string identifier system for registering functionality would, depending on the implementation method, also enable one plug-in to overwrite functionality provided by another one, by registering it later using the same name. However, the details of any implementation would need to be determined based on any limitations imposed by the implementation method itself.

For illustrative purposes, one may consider Python where a plug-in may register functionality using a string identifier within a dictionary maintained by the plug-in API. Figure 8 seeks to illustrate this, and exists only for illustrative purposes. Implementations using other languages would need to be tailored specifically to those other languages.

```
import logging

class PluginAPI(object):

    def __init__(self, app):
        self.functionality = {}

    def register_functionality(self, name, method):
        self.functionality[name] = method

    def request_functionality(self, name):
        return self.functionality.get(name, self.dummy_functionality)

    def dummy_functionality(self, *argv, **kwargs):
        return None
```

Figure 8: Example functionality registration and retrieval using Python.

Additional functionality, such as deregistering functionality or unloading plug-ins during runtime, could also be implemented as deemed necessary.

4 CONCLUSIONS AND FURTHER DEVELOPMENT

The original objective of drafting a speculative initial overall design proposal for a modification management utility would appear to have been met successfully and the proposed design was successfully based on previous research into various topics to a reasonable extent. One could argue that the design should be somewhat reasonable, however, the conservative amount of detail within it may restrict the usefulness of it, although any details themselves might depend on the chosen implementation method, such as the programming language, frameworks and other tools or environments used. The proposed design would also appear to be somewhat in line with the views of Yourdon and Constantine (1979, pp.16-26), in that every function of the utility could be separated into its own module in the form of plug-ins. Plug-ins could help better facilitate development, as well (Ye and Kishida 2003; Haapaviita et al. 2013). The number of hard-coded references could be reduced, for example between the main application and plug-ins (Mayer 2003), but also perhaps between various plug-ins themselves. Furthermore, depending on the implementation method, every higher level plug-in could be further broken down into a reasonable number of sub-modules depending on the method of implementation and the complexity of a set of functionality. Interaction between modules should correspond to actual interaction between parts of the problem, for example when installing a mod, some data about both the game itself and the mods to be installed, such as file paths, may be needed.

The concept of a minimal application core surrounded by plug-ins, while it would appear to work on paper, may require additional thinking and design effort if implemented in an actual piece of software. The separation of any and all functionality into plug-ins, that is, not only additional extensions but also core components that form basis of the utility, such as a user interface, filesystem abstraction, platform abstraction and the like, may also impose various risks in case the plug-in system somehow proves insufficient as a basis of a larger scale software solution. For a smaller scale modification management utility, however, the design may work, however it has not properly been tested yet. Overall, the design itself should facilitate reasonable development of a modification management utility as an open-source software project, based on the research used as guidelines. However, the proposed design specifically within the form proposed herein has not been extensively tested to the knowledge of the author, so it may be purposeful to collect data on similar designs and their usefulness before implementing the design in an actual software project.

However, even if the proposed new design—or any new design—would make sense with regards to maintainability, extendability and every other reasonable aspects to be considered, arguments could be produced against a complete redesign and rewrite of existing tools. For example, Wrye Bash

works, it has been successfully expanded to support a number of (similar) games and a complete rewrite could potentially experience issues with regards to regression in the form of bugs or absent features. Additionally, rewriting the utility from nothing back to its current feature level—even if no bugs were produced—could have the potential to consume expertise, time and effort that would be away from further developing the utility, just as with a gradual refactoring and restructuring project. There exist examples of rewrite projects being unsuccessful (Spolsky 2000), so a complete rewrite could potentially impose real risks. Rewriting a utility should perhaps not be made for the sake of rewriting only, and even with the source code structure of Wrye Bash being perhaps somewhat non-optimal, the current developers are still capable of working with it and further developing it.

Plans for further development of the proposed structure may include producing a prototype to test how the design scales. With the design concentrating on modularity and easy extension, producing a smaller scale prototype could allow for later expansion of the prototype itself into the final piece of software, depending on the prototype implementation, by porting or recreating functionality present within other utilities such as Wrye Bash. The design itself could also potentially benefit from various enhancements, perhaps, and practical implementation would most likely see the design adjusted to fit a specific development process, toolchain, language or platform.

5 SOURCES

1. Scacchi W. (2011) Modding as an Open Source Approach to Extending Computer Game Systems. In: Hissam S.A., Russo B., de Mendonça Neto M.G., Kon F. (eds) Open Source Systems: Grounding Research. OSS 2011. IFIP Advances in Information and Communication Technology, vol 365. Springer, Berlin, Heidelberg
2. van der Hoek A., Lopez N. (2011). A Design Perspective on Modularity. AOSD '11 Proceedings of the tenth international conference on Aspect-oriented software development. 265-280
3. Ye Y., Kishida K. (2003). Toward an understanding of the motivation of open source software developers. 25th International Conference on Software Engineering (pp. 419429). IEEE.
4. Yourdon E., Constantine L. (1979). Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Yourdon Press.
5. Lee, J. H., Jett J., Perti A. (2015). The Problem of “Additional Content” in Video Games. JCDL '15 Proceedings of the 15th ACM/IEEE-CS Joint Conference on Digital Libraries (pp. 237-240). ACM.
6. Haapaviita N., Jokelainen J., Reijonen P., Haikonen, J. (2013). Open Source Software Project Success Factors / Modularity As a Success Factor, University of Oulu, accessed 28 August 2017, <https://wiki.oulu.fi/download/attachments/34308420/ossd_2013_haapaviita_jokelainen_reijonen_haikonen.pdf>.
7. Lee J. (1997). Design Rationale Systems: Understanding the Issues. IEEE Expert: Intelligent Systems and Their Applications 12,3: 1997, 78-85.
8. Singer A.B. (2017) Plugins. In: Practical C++ Design. Apress, Berkeley, CA, Chapter 7: Plugins
9. Mayer J., Melzer I., Schweiggert F. (2003) Lightweight Plug-In-Based Application Development. In: Aksit M., Mezini M., Unland R. (eds) Objects, Components, Architectures, Services, and Applications for a Networked World. NODe 2002. Lecture Notes in Computer Science, vol 2591. Springer, Berlin, Heidelberg
10. Parnas, D.L.: Designing Software for Ease of Extension and Contraction. IEEE Trans. Software Engineering SE-5(2), 128–138 (1979)
11. Peng G., Geng X., Lin L. (2012). Modularity and Inequality of Code Contribution in Open Source Software Development. 45th Hawaii International Conference on System Science (HICSS). Proceedings. (pp. 45054514). IEEE.
12. *Wrye Bash source code repository at GitHub* [online]. (2017). Available from: <https://github.com/wrye-bash/wrye-bash>. [Accessed 28 August 2017].

13. *Mod Organizer source code repository at GitHub* [online]. (2017). Available from: <https://github.com/TanninOne/modorganizer>. [Accessed 28 August 2017].

14. *Vortex source code repository at GitHub* [online]. (2018). Available from: <https://github.com/Nexus-Mods/Vortex>. [Accessed 27 March 2018].

15. *'Contributing to Wrye Bash' at Wrye Bash GitHub repository* [online]. (2017). Available from: <https://github.com/wrye-bash/wrye-bash/blob/dev/Contributing.md>. [Accessed 25 August 2017].

16. *Issue 200 at Wrye Bash GitHub repository* [online]. (2017). Available from: <https://github.com/wrye-bash/wrye-bash/issues/200>. [Accessed 28 August 2017].

17. *'De-wx'ing', GUI APIs projects for Wrye Bash at Wrye Bash GitHub repository* [online]. (2017). Available from: <https://github.com/wrye-bash/wrye-bash/issues/190>. [Accessed 23 August 2017].

18. *'Deciphering Source Filenames' at Wrye Bash GitHub repository* [online]. (2017). Available from: <https://github.com/wrye-bash/wrye-bash/wiki/[dev]-Deciphering-Source-Filenames>. [Accessed 3 September 2017].

19. *'Should I use Python 2 or Python 3 for my development activity?' at Python Wiki* [online]. (2017). Available from: <https://wiki.python.org/moin/Python2orPython3>. [Accessed 23 August 2017].

20. Spolsky, J. (2000). *'Things You Should Never Do, Part I', Joel on Software* [online]. Available from: <https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/>. [Accessed 25 August 2017].

21. Utumno (2015). *'306 Code refactoring, performance' at Wrye Bash GitHub repository* [online]. Available from: <https://github.com/wrye-bash/wrye-bash/commit/1cd839fadbf4b7338b1c12457f601066b39d1929>. [Accessed 28 August 2017].

22. *'Wrye Bash General Readme' at Wrye Bash GitHub repository* [online]. (2017). Available from: <http://wrye-bash.github.io/docs/Wrye Bash General Readme.html>. [Accessed 28 August 2017].

23. *'Wrye Bash Version History' at Wrye Bash Github repository* [online]. (2017). Available from: <https://github.com/wrye-bash/wrye-bash/blob/dev/Mopy/Docs/Wrye Bash Version History.html>. [Accessed 23 August 2017].

24. *wxPython Wiki* [online]. (2017). Available from: <https://wiki.wxpython.org/>. [Accessed 3 September 2017].

25. *'Running Wrye Bash on WINE (Arch Linux)' at Wrye Bash GitHub repository* [online]. (2017). Available from: <https://github.com/wrye-bash/wrye-bash/wiki/[dev]-Running-Wrye-Bash-on-WINE-(Arch-Linux)>. [Accessed 3 September 2017].