Facultat d'Informàtica de Barcelona (FIB)
Barcelona School of Informatics

Universitat Politècnica de Catalunya (UPC)
BarcelonaTech

GL-Get: A dependency manager for software projects

Jordi Vilalta Prat

September 2016

Director: Lluís Solano Albajes
Department of Computer science (CS)

Informatics Engineering (2003)

Thanks

This is the end of a very important phase for me,
and it wouldn't have been possible for me to reach it
without the help of many great people.

Erica and little Aina, who help me wake up every day.
My parents for their unconditional support.

My coworkers in general, for making me want to go to work everyday...
even when you ask me "How's GL-Get going? Already finished?"
The GL-Get team: Albert, Jeremy and Alex (just pish to master!)
And the rest of the company: you rock!

Table of contents

Table of figures

Table of tables

Part 1. Introduction

1. Preamble

1.1. Context

The videogames industry has been one of the fastest growing for the last years, getting really close in revenue to other entertainment industries like cinema. But competitivity is also increasing, forcing companies to find new areas to expand, and ways to reduce costs.

Gameloft [1] is one of the leader companies worldwide in the mobile videogames industry. Since it was founded in 2000, it has expanded to having more than 5000 developers in more than 20 studios. It started by developing games for feature phones, but quickly jumped to the smartphones and tablets boat. It has created very successful franchises like Asphalt, Modern Combat, Order and Chaos or N.O.V.A., but it has also created games based on existing licenses, like Ice Age, Spiderman, Prince of Persia, Assassin's Creed, CSI, etc.

The Barcelona studio of Gameloft has been alive for more than 10 years and it has released several big quality titles: Asphalt 8, Despicable Me: Minion Rush, The Adventures of Tintin and Six Guns, to name a few. It's also been a pioneer on the so-called «freemium» model, by releasing the former 2 titles as free-to-play with in-app purchases for extra goods, making them the most successful games of the company to date.

Figure 1: Some of the videogames developed in Gameloft Barcelona: Asphalt 8, Despicable Me: Minion Rush, The Adventures of Tintin and Six Guns

In order to continue being competitive, the Barcelona studio is working on redefining processes and creating new tools to improve the development work-flow, while having an eye on future projects. The mid-long term goal is to reduce costs while improving the quality and maintainability of the developed games.

1.2. Motivation

The development process for triple-A videogames, even for the mobile ones, is very long and complex, and it involves lots of different areas: artists, programmers, game designers, producers, etc. Working on a videogame usually involves handling big amounts of data (tens of code libraries and hundreds of gigabytes of assets).

Videogames usually do intensive usage of the device's resources, so they're developed using native code (mainly C++) in order to take profit of each CPU cycle. In addition, there's currently many different platforms for mobile devices (iOS [2], Android [3], Windows Phone, Tizen [4], ...), so the code has to be multi-platform. Managing this kind of projects is an important task, and unfortunately the available tools nowadays aren't very complete or they're for a very specific environment.

A task as simple as keeping the project files up to date and in a compilable status becomes tedious when the multi-platform factor comes in: some platforms require the usage of specific tools, and even full IDEs in order to generate its binaries. To name a few examples, in order to build for iOS you must use Xcode on OS X, in order to build for Windows Phone you must use Microsoft Visual Studio on Windows, etc. In order to solve this issue there are some tools that can generate the project files required for each platform by taking a file that contains a project description. CMake [5], qmake [6] and Premake [7] are some of the most known tools for this job. In the case of Gameloft Barcelona, Premake is used for this job.

Unfortunately this is just a small example of the difficulties found when having to manage big software projects. Everything gets much more complex when sharing code among several projects, and even more if it happens across several studios, like in the case of Gameloft.

Building code libraries is a very nice way to reuse code, but having the code of the library separated from the user code means that when there's a change in the library, all the projects using it have to update. If there's no kind of version locking, the code will simply break and stop building. Managing a small number of dependencies could be done by hand, but it's a chore and it quickly becomes a source for human errors. For that we need a dependency manager tool, which can make this process as simple as possible.

During the last few years some projects have emerged aiming at creating a C++ dependency manager: conan [8], biicode [9], CPM [10], Pacm [11], CVM [12], and probably some more. Unfortunately, many of these were just experiments or have been abandoned at the time of writing. In addition none of these targeted Premake as its project generation system, so it would make it harder to reuse the customization work already done at Gameloft. For these reasons it was decided to create a new custom tool tailored at the company's specific needs.

1.3. Objectives

The project objective is the development of a multi-platform command line tool that helps in the development of software applications. Its main task will be the management of software dependencies, but at the same time it will offer several complementary features that will help in the videogames and libraries developers' work-flow.

Some of these extra features could be build automation, unit test execution, version management, partial library downloads (sources or binaries for a given platform, depending on whether we want to work on a library or just use it as a dependency), management of shared folders where packages used in several projects can be stored, etc.

This new tool should also integrate well with the current work environment and tools used at the company, so the costs of adoption can be reduced to a minimum.

As a consequence, the usage of this tool should help the company achieving some objectives:

- Ease the creation, usage and management of software libraries
- Improve and optimize the development work-flow
- Process automation and human error reduction

In addition, some of the objectives for the tool were not defined at the moment of starting the project, so part of the job is working closely with the rest of the team (the people considered to be the clients of this project) to identify the rest of requirements or improvement points during the development process.

1.4. Document structure

This document is split in 3 main parts: Introduction, Development and Annexes.

The Introduction describes the context of the project, what are the motivations for it and the objectives. It also contains a brief history on the project's origins and the team working on it, and an explanation on a few of its implementation decisions.

The Development is the main part of this document. It covers the processes of specifications gathering and analysis, design, and the implementation details. At the end there's also the planning and cost analysis and the conclusions.

The annexes contain big pieces of text that are part of the work or need to be referenced from other parts of this document, but that would break the flow of the document if inlined. A few examples are the user manual for the final application and the reference of the command line options for the application.

2. Beginnings

This project is developed inside the Gameloft [1] company, where the author was already working. This section describes the starting point for this project.

2.1. Initial status

The application development had already been started but it was in a proof of concept stage and it was being developed on the spare time of a few co-workers.

That initial stage was developed as a Premake [7] extension. Premake is the project file generation tool used in Gameloft Barcelona. It has a small core written in C, but it's designed to be extensible using the Lua [13] programming language, and most of its functionality is written as Lua modules. Given the big amount of in-house Premake experience, its easy extensibility and the quick development cycle thanks to most of its functionality being interpreted and not having to recompile to test, it seemed like the best option for quickly prototyping the desired functionality.

2.2. Team

The people who had been working on the proof of concept couldn't continue developing it because they had to focus on other projects, so short before the beginning of this project, a small team was created to formalize the development: one of the people who worked on the proof of concept was chosen as the project lead, and two more co-workers plus the author joined the team as full time developers.

Having a team of 4 people (actually 3 full time developers) working on the same project would push the development of the tool, but we would also have to coordinate what parts would everyone work on. At the beginning we just had a few small tasks, so we used a small online tool for task management and tracking called Asana [14] to get organized. We started each picking the next task to work on from the backlog of pending tasks. Later on we started each focusing on different parts of the development.

2.3. Programming languages

Since the proof of concept was already implemented in Lua as a Premake extension, we took that as a base for our work and continued implementing most of the functionality in Lua. All of us came from the C/C++ world and we didn't have much experience with Lua, so at the beginning we had to learn it and the first tasks served as a learning exercise.

At some points, where we tried to push the limits of Lua, we had to jump back to C or C++ in order to plug native functionality to the Lua VM.

2.3. Programming languages

PART 2. DEVELOPMENT

3. Specification

While the high level requirements were known beforehand, many details weren't specified until we faced new doubts in the course of the implementation. The requirements detailed here are a compilation after having run the project for some time.

3.1. Problem

3.1.1. Sharing code

Before starting this project, the whole development in the company relied on Subversion [15] (SVN). Both source code and data was stored in a centralized Subversion repository in a fairly monolithic way. There was an attempt to modularize projects: code was being split into libraries, which were located in separate folders of the same repository. Thanks to Subversion's externals [16] method, one could build a single working copy that included checkouts of other repositories in sub-folders of the main checkout. This allowed each library to specify their own dependencies and have local references to them (Figure 2). In practice, it would lead to a lot of duplicated checkouts when several libraries shared the same dependencies (Figure 3).

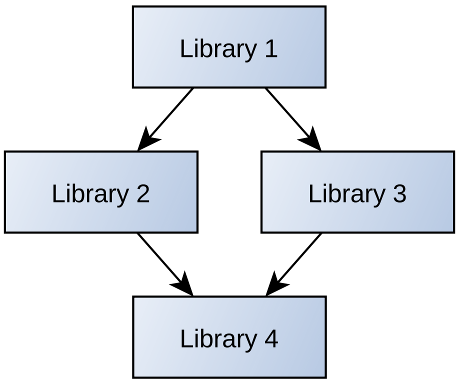


Figure 2: Logical library hierarchy

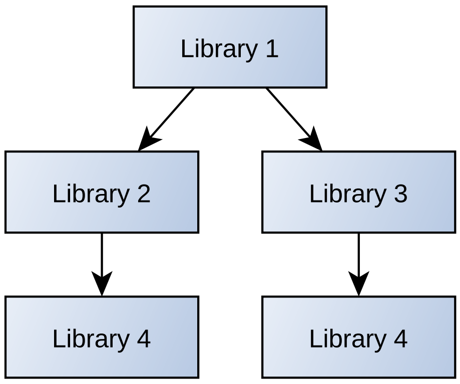


Figure 3: Physical checkout hierarchy

It ended up wasting lots of hard drive space and network bandwidth, especially considering that one of the lowest level and most shared libraries was Boost [17], which used around 3Gb for each checkout. In addition, when working on several libraries at the same time, changes wouldn't be synchronized between checkouts of the same library, leading to confusing compilation errors.

In the end this approach proved impractical, and it was decided to put all the libraries together on the same folder, either directly or as externals to their own repositories, so the checkouts could be shared, as shown in Figure 4. Libraries had references to their dependencies in the upper directory.

While it worked, it had several drawbacks. One always had to do a full checkout of the libraries repository, even if just wanting to use a few of them. The versions of the dependent libraries couldn't be specified per project, they were determined by the parent folder which contained them. All the projects contained on the same repository had to share the same versions of their dependencies. Its main advantage (removing the dependency management) essentially returned the monolithic feeling and lack of flexibility.

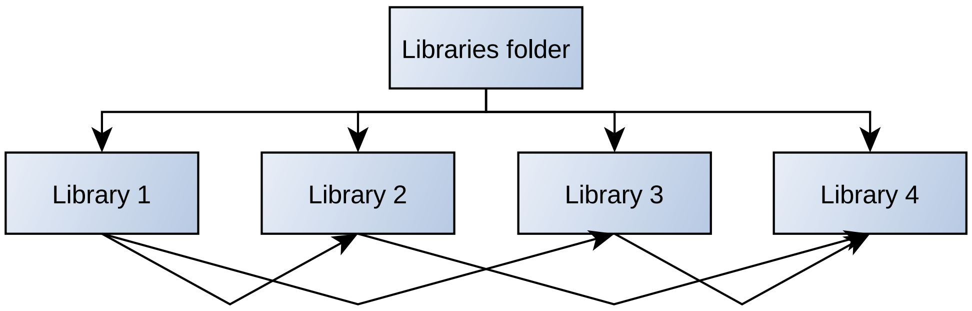


Figure 4: Shared libraries folder

3.1.2. Versioning

While keeping all projects on the same folder allowed them to be always in a consistent state, it limited the development to a linear fashion. Subversion itself allowed branch and tag creation, but the way the repositories were organized made it harder to use.

In the end some projects ended up collecting sets of patches with project-specific functionality, or with fixes that were pending approval from the library owners. These patches were applied on the local computer of each developer after doing the checkout, making it harder to update, and it was a complete chore to update the patches to newer versions of the original library. Additionally, if these patches were ever adopted by the library owners, they would only be released in a new version in the main development line, making it mandatory to update to the latest release, which would probably break the API compatibility due to the accumulated changes (Figure 5).

Introducing branches and release tags in the work-flow would allow these patches to be applied on a branch at any point of the project history and release hotfix releases (Figure 6) that would keep the API compatibility.

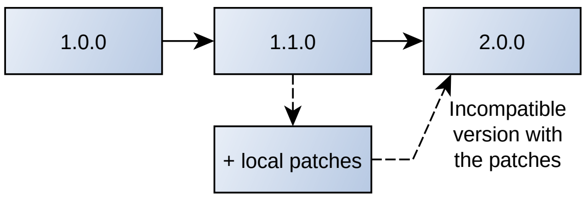


Figure 5: In lineal development, each project has to keep its own local patches

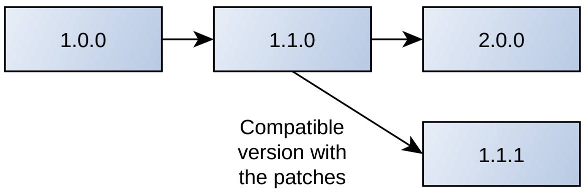


Figure 6: In branched development, the library owner can release hotfixes of previous releases

Having a good versioning system would isolate the development line of each library, letting the authors of each library decide when to introduce breaking changes, keeping the option of coming back to an earlier release and adding the necessary fixes to support users of those versions. It would also allow sharing of those fixes among projects instead of being exclusively available to the owner of the patch.

3.1.3. Project file creation

Targeting many different platforms sometimes enforces using specific toolchains that require project files made explicitly for them. Doing it manually would be a lot of work and it would be an error-prone process (it's easy to miss changes while synchronizing platforms).

Fortunately there are already good tools to handle this problem. Premake [7] is one of them, and it was already adopted before the start of this project. Premake allows the user to create scripts (usually called premake5.lua) that describe software projects in a platform-independent way. A typical basic Premake script would look like this:

```
workspace "HelloWorld"
  configurations { "Debug", "Release" }

project "HelloWorld"
  kind "ConsoleApp"
  language "C"
  targetdir "bin/%{cfg.buildcfg}"

  files { "**.h", "**.c" }

  filter "configurations:Debug"
    defines { "DEBUG" }
    flags { "Symbols" }

  filter "configurations:Release"
    defines { "NDEBUG" }
    optimize "On"
```

Premake takes an "action" as its first argument. The action is the exporter to be used, typically the name of the target platform that has to be generated. Premake has built-in support for Microsoft Visual Studio, makefiles, Xcode, MonoDevelop, and a few others. It supports extension modules that can add, among other things, support for exporters and several are available, like support for Android and CMake.

Running "premake vs2013" on the folder that contains this script would generate project files for Visual Studio 2013 (HelloWorld.sln and HelloWorld.vcxproj), that could be opened straight away and would create a HelloWorld.exe binary on the bin/Debug or bin/Release folder based on the chosen configuration.

Running "premake gmake" would generate a set of makefiles that would allow building the same project on Linux, having a lot of platform specific particularities in place, like the binary name, or compiler flags, while keeping the options specified by the user, like the target folder or the configuration defines.

3.1. Problem

Thanks to that, and combined with a simple batch file that uses Premake for each desired target, every time the project files or project options are changed, running the batch file makes sure all the project files are up to date (Figure 7).

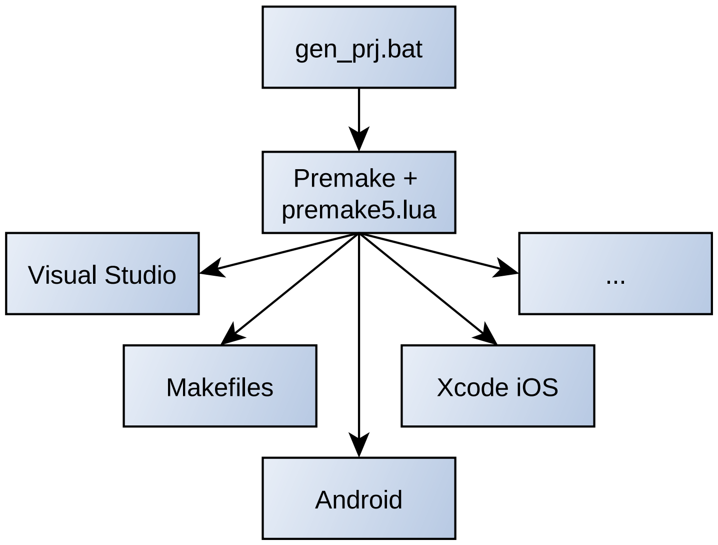


Figure 7: Regenerating the project files for all the targets is as easy as running a batch file.

While this works very well for simple projects with few dependencies, projects with more dependencies get harder to configure very soon. When a project has to link to a library, the actual library and the library search path have to be specified:

```
project "HelloWorld"
  libdirs { "../library1/bin" }
  links { "library1" }
```

Premake on its own doesn't know how to extract information from the included projects, so when a dependency has its own dependencies (transitive dependencies), these have to manually be added to the Premake script to make the linking work. For instance, if library1 depends on library2, and library2 depends on library3, the Premake file for the HelloWorld project would look like this:

```
project "HelloWorld"
  libdirs { "../library1/bin", "../library2/bin", "../library3/bin" }
  links { "library1", "library2", "library3" }
```

Changing the required dependencies of library3 would make the library2, library1 and HelloWorld projects fail to build, having to go one by one fixing the linking options, which aren't always trivial if you're not aware of the latest changes of the transitive dependencies. Having a dependency manager in place could make it easier by automating the link data injection, deriving it from the package information.

Other than that, the Premake version used in Gameloft includes many customizations to improve project portability, to add support for several non-standard platforms and to adapt it to the company's needs and conventions. Managing these customizations is often done in an ad-hoc way, which makes it hard to track which is the required version of Premake, and it's easy to add compatibility breaks. Having our own branch of Premake would allow us to make our own releases and control these customizations in an easier way.

3.1.4. Continuous integration

In order to keep track of the software quality it's convenient to use continuous integration. It consists of a set of processes that execute every time a change is detected in a software package. It usually involves checking out the repository, building the package, building its tests and run the test suites, checking the results for regressions.

When it's well configured, it's a very valuable resource and it can help the developers by running the tests on platforms they don't own and notifying them when the status of the software package changes. The main problem with it is that the configuration process is often manual and non-trivial, and it has to be done per package. This leads to some packages being forgotten and not benefiting of continuous integration.

3.2. Analysis of the existing tools

It's important to get an idea of the implications of developing a dependency manager. Analyzing the existing tools will also help understanding the features that the users will expect.

There's a very insightful article by Sam Boyer called "So you want to write a package manager" [18] which covers a lot of the areas of package manager implementation. Based on it, package managers can be categorized in:

a) OS/system package manager (SPM)

b) Language package manager (LPM)

c) Project/application dependency manager (PDM)

The main distinction between them is the main users of the system: SPMs are meant to help users installing software packages, PDMs are aimed to help software developers to create new software, and LPMs are often a mix of both.

The tool to develop will include some extra features in addition to the package management. Once the source packages are in place, it will integrate with the other main tools involved in the process: project file creators and build tools.

The area of build tools is much more open and often imposed by each target platform.

3.2.1. System package managers

System package managers are in charge of managing the applications and libraries installed in an operating system for use of its users.

It's not mandatory for an operating system to have a package manager but it's a common feature nowadays and it makes the life of its users much easier, specially when dealing with application and system updates. It also allows saving disk space, since having a single central package manager allows libraries to be physically shared between applications. Usually each operating system has its own package manager, but some are available for several operating systems.

Table 1 offers a small list of the main available ones. There are many system package managers with a very wide spectrum of features. Many of them have dependency tracking and some even allow choosing if we want to build the package from its sources in order to tweak it for the target machine. At the opposite end, mostly for commercial systems, the package managers don't track dependencies and consider each package as an atomic binary unit that can run on its own. In this case their functionality is almost reduced to a centralized catalog of software and update notification.

Name	Operating Systems	Source/Binary	Manages dependencies
Apt [21]	Debian Linux	Source+Binary	Yes
RPM [20]	Red Hat Linux	Source+Binary	Yes
Pacman [25]	Arch Linux	Binary	Yes
Portage [27]	Gentoo Linux	Source+Binary	Yes
Nix [19]	Linux, OS X	Source+Binary	Yes
Zero install [33]	Linux, OS X, Windows	Source+Binary	Yes
FreeBSD Ports [36]	FreeBSD	Source+Binary	Yes
pkgsrc [24]	*BSD, Linux, OS X, Solaris, ...	Source+Binary	Yes
pkgutil [26]	Solaris	Binary	Yes
Homebrew [29]	OS X	Source+Binary	Yes
Fink [22]	OS X	Source+Binary	Yes
MacPorts [28]	OS X	Source+Binary	Yes
Mac App Store [35]	OS X	Binary	No
Windows Store [34]	Windows	Binary	No
Chocolatey [31]	Windows	Binary	Yes
App Store [30]	IOS	Binary	No
Google Play [32]	Android	Binary	No
PlayStation Store [40]	PS3, PS4, PS Vita	Binary	No
Xbox Games Store [39]	Xbox 360, Xbox One, Windows	Binary	No
Nintendo eShop [37][38]	Wii U, 3DS	Binary	No
Steam [23]	Windows, OS X, Linux	Binary	No

Table 1: Comparison of system package managers

3.2.2. Language package managers

Language package managers are in charge of retrieving, building and installing packages of a concrete programming language in a way that can be shared among all other installed software in the machine. The advantages of being specific to a given language are that it can take advantage of the language specifics, and it can provide higher integration levels for the packages than a generic package manager. They're usually tied to the install directory of the language toolchain.

Usually each programming language provides its own package manager and a repository of software ready to be used. Table 2 offers an overview of some of the available LPMs.

Language	Repositories	Tools
C / C++ [68]	Many	conan [8], biicode [9], CPM [10], Pacm [11], CVM [12], Bazel [89], CoApp [90]
C# [42]	NuGet [62]	
Common Lisp [80]	Quicklisp [76]	
Go [41]	Go packages [79]	go tool [67]
		godep [61]
		Glide [60]
Haskell [66]	Hackage [65]	cabal [59]
Java [55]	Maven central [83]	Maven [58], Ivy [81], Gradle [87]
JavaScript [77]	npm [47] for Node.js [84]	
Lua [13]	LuaRocks [72]	
	LuaDist [64]	
Objective-C [78]	CocoaPods [75]	
Swift [57]	Swift Package Manager [85]	
Perl [54]	CPAN [53]	cpan [52], cpanm [82], cpanp [86], local::lib [88]
	PAR [46] PPM Index [74]	ppm [45]
PHP [44]	PEAR [51]	pear
	PECL [50]	
	Packagist [73]	Composer [71]
Python [63]	PyPI [49]	pip [43]
		Buildout [70]
	Conda [56]	
Ruby [48]	RubyGems [69]	

Table 2: Available language package managers

3.2.3. Project dependency managers

While LPMs could be enough for some use cases, most of them just offer a common pool of packages and miss some features required by developers or for managing complex projects.

As an example, some LPMs just allow installing a single version of a given package, which limits some features, like having installed or be developing several packages that need different versions of the same dependency.

PDMs concentrate on a single project and usually install the dependencies in a location isolated from any other projects.

3.2.4. Project file creators

As noted previously, the decision for the project file creator was already taken in advance: Premake is already used in the company and it covers most of the required features, and the users are already familiar with it.

In addition it's very extensible, so it will make it easy to integrate with the new tool.

3.2.5. Build tools

There's a lot of build tools available. Some platforms have a customized build tool and are very restrictive on the platform they can be run, but others can be used for several target platforms depending on the compiler it's configured to use, so the final usage will widely depend on the concrete use cases.

Table 3 shows some of the common build tools used at Gameloft to build games and tools using C++ code natively.

Build tool	**Host platform**	**Target platform**
MSBuild [93]	Windows	Windows, Windows Phone (in VS2015 also Android and iOS)
Xcode [92]	OS X	OS X, iOS, tvOS
Gradle / Ant [94]	Windows, Linux, OS X	Android
Make [91]	Windows, Linux, OS X	Windows, Linux, OS X

Table 3: Common C++ build tools

There are also lots of other build tools targeting other programming languages, but these are out of the scope of this project. Maven is a good example of a tool that integrates both dependency management and building for applications written using the Java programming language.

3.2.6. Continuous integration tools

There's also a wide variety of tools for continuous integration. The one being used at Gameloft Barcelona is Jenkins [95], which in addition to being free and open source, is very widely spread and has lots of extension plug-ins available. It's easy to find plug-ins that add very interesting features to it: version control systems, compilers, test result analytics, build statistics, or even release deployments.

3.3. Requirements analysis

Based on the analysis of the previous sections and conversations with the project owners, we've extracted a list of requirements, both functional and non-functional, that the system must fulfill.

3.3.1. Functional requirements

1. **The system must be able to know all the information it needs from a project by reading a single file.**

   Each project should be accompanied by a file (we'll call it manifest) that contains all the information that the system may need to know about it.

2. **The system must know which versions of a project are compatible.**

   Given the references to two different versions of the same package, the system must be able to know if these are compatible. This basically means that if a project was developed using one version of a dependency, it should work just fine by replacing the dependency with the compatible version.

3. **The system must be able to resolve the list of dependencies of a project.**

   Taking the project information as a base, it should be able to get the information of the dependent projects (both direct and transitive), and it should build a list of all its dependencies and which version of each should be used.

4. **The system must manage a folder where the shared projects will be located.**

   Projects that can be shared as dependencies of several other projects at once should be downloaded on a common folder. The system should be able to manage the available projects.

5. **The system must be able to decide where each of the dependencies should be located on the disk.**

   Based on the projects information and the relationship between them, the system must be able to determine where each project should be located on the disk. For instance, some of the projects should be located in a shared folder, so other projects can use them.

6. **The system must be able to download a software project and its dependencies.**

   Running a single action, the system should be able to download the specified project and all of its required dependencies from the VCS so the user can start working on it right away.

7. **The system must support different types of VCSs.**

   Software projects are hosted using Version Control Systems. While Gameloft is using majorly Subversion, there are plans to start using Git for some new projects, so sooner or later the system should support at least these two.

8. **The system must be able to download packages from arbitrary locations.**

   The origins of the software projects should be open to arbitrary values given by the users, who may need to get projects from locations unknown at the time of the system creation. The system has to support several project repositories and even upstream projects which aren't aware of the system and its conventions. The available repositories must be configurable.

9. **The system must know how to download a project given a VCS location or a project description.**

   The user should be able to specify arbitrary dependencies by using the concrete VCS location. To simplify the most common cases, there should be some default repositories configured where the projects can be looked for, by just knowing their name.

10. **The system must manage VCS credentials.**

    When working with projects having dependencies coming from different repositories it may become annoying to do a simple update, since other systems could ask for many passwords to the user. Since the system should also be able to list the contents of remote repositories in order to know the available projects or their versions in advance, it would easily become unusable if it had to request a password to the user for every action. The system must have a way to specify these credentials in a simple way so it doesn't bother the user.

11. **The system must be able to work on a project that wasn't downloaded.**

    Sometimes the users may want to create a project by hand or create a new one by copy-pasting the contents of another one. In these cases, the project folder isn't associated to a VCS. The system should be able to work with these situations and manage its dependencies anyway.

12. **The system must handle projects that have companion or helper projects very closely related to them.**

    Most software projects aren't developed completely alone. Most of them are accompanied by test projects, or in some cases by tools, samples or even internal libraries. These complex projects should be handled as a whole, allowing the user to easily work on all of these at once.

13. **The system must allow the user to easily work on several related projects at once.**

    When working on library projects, developers often want to try how the changes affect other projects that use them. In some cases this could even be a set of related libraries that have to be developed together, but they're not close enough to be considered part of a whole. In these cases the system should allow the user to work on these projects together.

14. **The system must be able to use projects which aren't aware of it.**

    The user may need to use an upstream dependency, which comes directly from its author outside the company, and it may not have the expected structure or the information files. The system should allow the user to provide the required information for these projects in a way that they can be integrated as normal dependencies.

15. **The system must give reproducible and consistent results.**

    The projects developed using the system will be committed to a VCS and will be used by a build machine and also by different developers in a different computers. Running the same commands on the same project version should give the exact same results on different configurations unless explicitly requested by the user. That includes the computation of the resolved dependencies, the contents of the generated project files, and the targets generated by compiling the projects, among others.

16. **The system must be able to create project files for multiple platforms.**

    The developers use at least Microsoft Visual Studio and Apple's Xcode as IDEs for development, and the company builds software for Microsoft Windows, the UWP (Universal Windows Platform for Windows 10 and up), Windows Phone, Linux, Mac OS X, iOS, watchOS, tvOS and Android, so at least these platforms must be supported.

17. **The system must simplify the task of specifying libraries to link.**

    When generating the project files, the system should automatically derive the information of the libraries to link from the resolved dependencies. That way, the user only should worry about specifying the first level dependencies, and the system should take care of providing the rest so the build succeeds.

18. **The system must be able to build a project and its dependencies.**

    It should provide an easy way to build a project and everything it needs, that is dependencies, internal helper tools, test applications, etc. For it, it will need integration with some build tools.

19. **The system must be able to run the unit tests of a given project.**

    Unit tests are usually simple applications that don't need user interaction and return a single value that indicates if the tests passed or failed. Following these simple conventions, the system should be able to run test applications and verify the status of the project based on it.

20. **The system must download just the required parts of the projects.**

    Projects may be grouped with several secondary parts (tests, tools, compiled binaries, etc), and in some situations not all of these are needed. When it's not necessary and if the user doesn't force it, the system should try to use the minimal amount of bandwidth and disk space required for the project to be functional.

21. **The system must provide commands to modify the project information.**

    Adding or removing dependencies, changing the project structure, and other tasks may be a bit complex if the user has to manually modify the files that contain the project information. The system should provide some commands that guide the user in this process.

22. **The system must help the users make their projects adhere to some conventions.**

    The project folder hierarchy or the target binaries file-name mangling are some of the things that should follow some common conventions. When everyone uses the same conventions, it's much easier to know what everything means when starting to work on someone else's project.

23. **The system must help the user work using branches.**

    Working with VCS branches can be a bit of a chore, specially when using Subversion. The system should do its best not to make the situation worse, and try to improve it as much as possible. If some common tasks for branch creation or merging can be automated, it should do it.

24. **The system must make it easy to create project releases.**

    The current team isn't used to work with releases. Since we want to introduce them in the way people work, the system must make it easy. It should for example contain a command to release a version, which makes several checks and commits the binaries for others to use.

25. **The system must help the user manage different project configurations.**

    In some cases configurations differ just by an optimization flag or a define, but in some other cases, configurations may add breaking changes to the ABI. In those cases, the system should take care of using the matching configurations of the dependencies and take all the needed actions when preparing the project files.

3.3.2. Non-functional requirements

1. **The system shall be easy to use, specially to people used to work with Premake.**

   Since the system is aimed at users who already know Premake, making it follow the same conventions or using the same terminology should help them start using and feel more familiar with it. Anyway the user interface should be as easy to use as possible for anyone.

2. **The system shall be self-documented.**

   Once the user knows the basics, he should be able to learn how to use new features by just using it, without having to read any external documentation.

3. **The system shall be easy to install.**

   If the system has any dependencies, these must be embedded or installed together. Ideally it should be distributed as a standalone binary with no external dependencies and no need to install it, being able to run it directly from the directory it was downloaded.

4. **The system shall be configurable.**

   The user must be able to change system settings using simple configuration files.

5. **The system shall log its actions to disk.**

   In addition to showing some progress information to the user, the system should log its actions to disk in order to help debugging.

6. **The system shall be portable.**

   The users that will be using the tool are using several of the major desktop operating systems (Windows, Linux and Mac OS X), so it should run at least on these.

7. **The system shall be efficient enough to manage projects with tenths of dependencies within reasonable time.**

   The projects that should be managed by the system may end up having around a hundred of dependencies, so the system should be able to manage them with relative efficiency. Some of the tasks that are done with less frequency may take more time, but tasks that could be performed several times per day by the developers should be quite efficient.

8. **The system shall try to reduce the used network bandwidth and disk space.**

   When working on several projects, a user may have several copies of the same dependencies on its hard drive, in addition to downloading updates more than once. The system should do its best in order to reduce the usage of these resources. For instance, the system should be able to run most of its tasks without network access once the projects the user wants to work on have been downloaded.

9. **The system shall be extensible.**

   It should allow the users to write their own custom commands by reusing the APIs of the internal modules, and it should be easy to add new functionality.

10. **The system shall be protected against malicious code hosted remotely.**

    The nature of the system lets it download files hosted in remote machines which may not be under control of the user. A malicious user might configure a package in a way that made the system crash. The system should be protected against this.

11. **The system shall be testable.**

    In order to make sure no regressions are introduced during the development, the system should be developed together with a test suite that checks its behavior.

3.4. Used tools

The following is a non-exhaustive list of the tools used during the development of this project:

- **Operating systems**

  Most of the development was done in Microsoft Windows (versions 7 and 10), but Linux and Mac OS X were also used for testing. Mobile operating systems like iOS and Android were used just for testing the resulting binaries.

- **Programming languages**

  Some parts of the application were done using C++ [68], but the largest part was implemented in Lua [13].

- **Version control systems**

  The code of the project was hosted using Git [96], but Subversion was also used for some of the dependencies. At a later stage GitLab [97] was added for Git repository browsing and management of branch merge requests.

- **Integrated development environments**

  The main IDE used during the development was Microsoft Visual Studio 2013 [98], with the addition of the babelua extension [99], which added Lua support to it. At some points Vim [100] and Notepad++ [101] were also used for code editing.

- **Build tools**

  Each platform required using specific build tools for it:

  - On Windows I used the compiler integrated in Visual Studio [98].
  - For Linux testing, the standard toolchain was used (GCC [102], make [91], ...)
  - Testing on Mac OS X required the usage of Xcode [92].

- **Debugging**

  To debug C++ code, the Visual Studio integrated debugger was used. Decoda [103] was used for Lua debugging (when it worked).

- **Project file generation**

  Premake [7] was used to generate the project files for multiple platforms.

- **Interpreters**

  Lua for Windows [104] was the chosen Lua distribution to run Lua scripts directly on Windows. In order to build the project, the standard Lua VM [13] was embedded.

- **Virtual machines**

  To support running tests that required access to remote machines or machine isolation in the test suite, we used Vagrant [105], with VirtualBox [106] as the back-end.

- **Continuous integration**

  Jenkins [95] was the chosen continuous integration tool, and it was installed using Docker [107].

- **Work tracking and documentation**

  During the development of the project Asana [14] was used for task management (though it was later changed to Jira [108]), and Confluence [109] was used to write documentation.

- **For writing this document**

  The main tool used for it was LibreOffice [110], with the addition of the yEd Graph Editor [111] to draw the diagrams, and Microsoft Project [112] for the Gantt diagram.

3.5. Use cases

Use cases help to understand the ways the users can interact with the system. The following sections do this analysis.

3.5.1. Actors

The application is unaware of the existence of different user roles. We could specify different actors, like a library developer, an application developer, a code reviewer, or even the build machine as different actors, but the system behaves exactly the same for all of them, and most of them may want to run the same use cases, so there's no difference in practice. For this, I will unify them all into a single actor, called generically "User".

3.5.2. Use case diagram

The use case diagram shows what are the available use cases and who triggers them. Since several use cases are reused internally by the application, drawing all of the use cases on a single diagram resulted in a crowded diagram with many crossing arrows, making it hard to understand. For this reason several smaller diagrams are presented (Figures 8, 9 and 10), grouping related use cases. Some use cases may appear on more than one diagram.

Additionally, Figure 11 is provided to help understand how the use cases fit in the common workflow. It shows a flowchart relating the most common use cases and how they transition between states. The most common states during work are marked.

3.5. Use cases

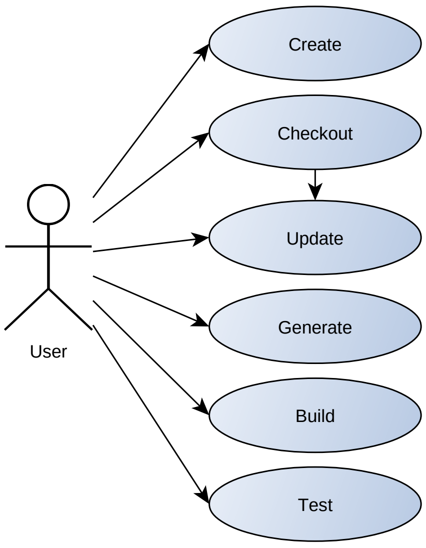


Figure 8: Basic use cases

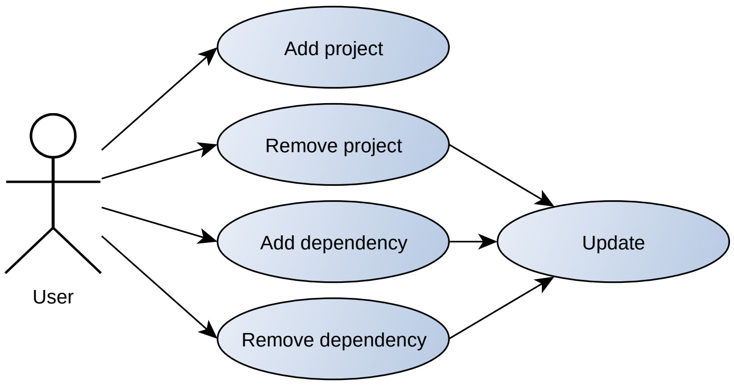


Figure 9: Structure management use cases

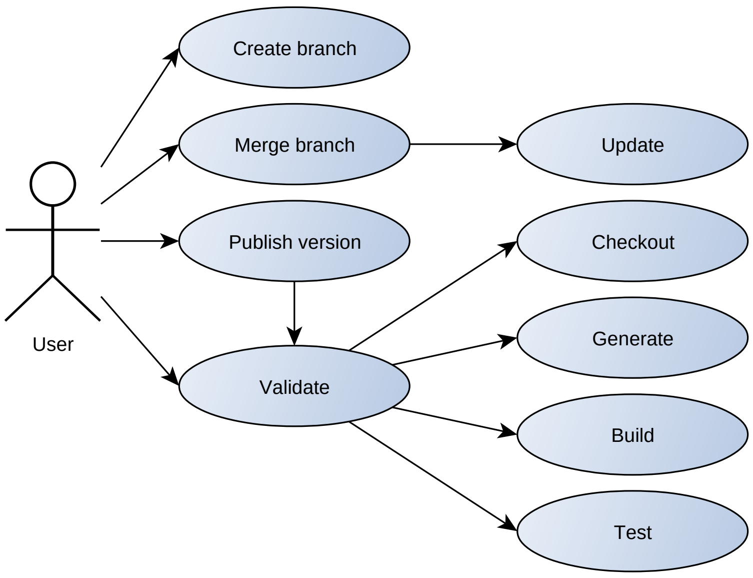


Figure 10: Release management use cases

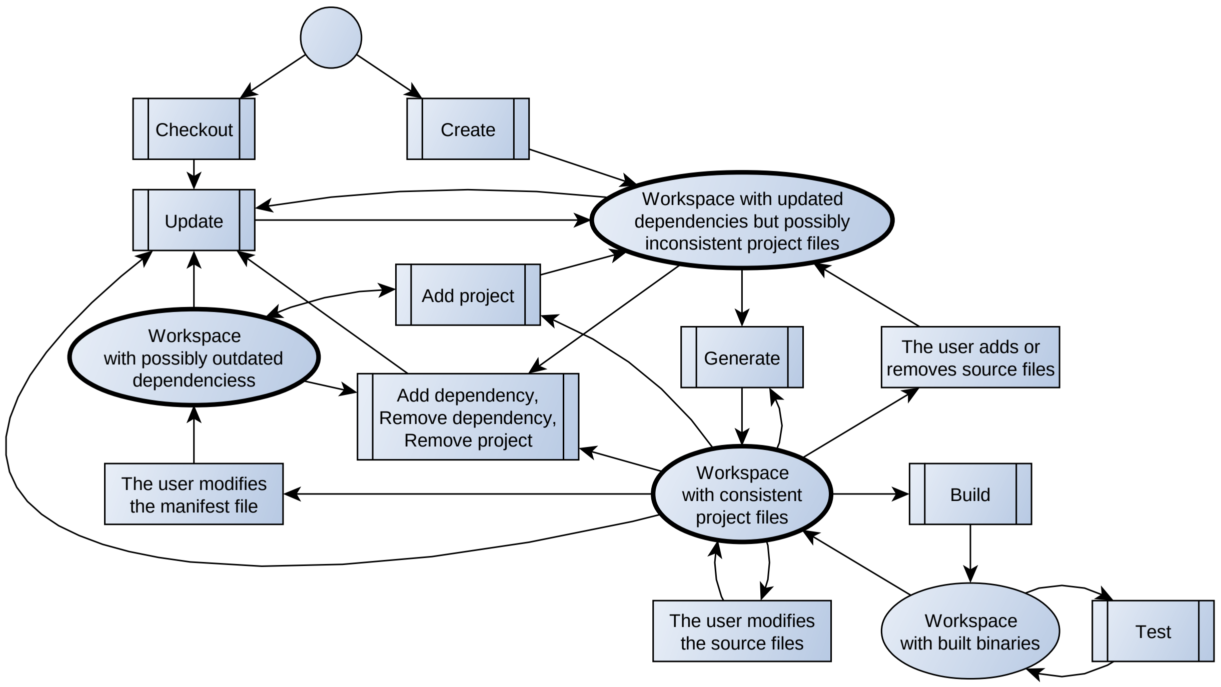

Checkout
Create
Update
Workspace with updated dependencies but possibly inconsistent project files
Add project
Workspace with possibly outdated dependenciess
Generate
The user adds or removes source files
Add dependency, Remove dependency, Remove project
Workspace with consistent project files
The user modifies the manifest file
Build
The user modifies the source files
Workspace with built binaries
Test

Figure 11: Use case flowchart

3.5.3. Use case description

This section details the system use cases together with some of their working conditions and their main steps.

Name	Create
Description	The user wants to create a new project to start working with it.
Trigger	The user.
Precondition	None.
Flow	1. The user can give the system the known project attributes via command line. 2. The system requests the missing information to the user. 3. The system creates the new empty project workspace, with the required folders and files to help the user start writing code. 4. If the user requested it, the system will upload the project to the VCS and will keep it as a working copy, ready to do commits and updates in the future.
Exceptions	• In case the data given by the user to the system is inconsistent, the system won't create the workspace.

3.5. Use cases

Name	Checkout
Description	The user wants to download a project already existing in a remote VCS and start working with it.
Trigger	The user. The validate use case.
Precondition	None.
Flow	1. The user specifies the information required to get the project. It could be the package name and its version in case it resides in a known repository, or the full URL in case it's in an arbitrary repository. 2. The system gets the manifest of the project from the VCS. 3. The system builds the basic workspace structure from the main package manifest. 4. The system jumps to the update use case to get the project contents and its dependencies.
Exceptions	• If the system can't access the specified project, it will exit without creating the workspace. • If the manifest of the specified project contains errors, the process will stop. • Further exceptions can raise during the execution of the update use case.

Name	Update
Description	The workspace and its dependencies must be updated to match what's specified in the manifest.
Trigger	The user. The checkout, remove project, add dependency, remove dependency or merge branch use cases.
Precondition	A workspace must exist.
Flow	1. The system reads the manifest of the main project of the workspace. 2. The system resolves the dependencies by downloading the manifests of the dependencies specified in the manifest and deducing what's the version that should be used. 3. The system will make sure each of the dependencies is downloaded to the path where it should be, and it will also update the ones that were already downloaded. 4. If there are issues checking out or updating any of the dependencies, these will be shown together at the end to let the user know the status.
Exceptions	• If the system can't access some of the specified dependencies, it will show an error to the user and it will stop the process. • If there's a conflict during the dependency resolution, an error will be shown to the user, telling it to resolve the conflict manually by updating the required dependencies versions, and the process will stop.

Name	Generate
Description	The user wants to generate the project files for the package in the workspace so it's ready to build or to open with an IDE.
Trigger	The user. The validate use case.
Precondition	A workspace must exist.
Flow	1. The user specifies a target toolset to generate. 2. The system builds the information that Premake needs about the projects and its dependencies for each package solution. 3. The system runs Premake on each of the solutions of the package, generating the project files for the requested toolset. 4. Optionally, the system does the same for all the dependencies, allowing the user to work on all of the dependencies with a single command.
Exceptions	• If the Premake scripts for a given project aren't configured properly or its information doesn't correspond to what's specified on the manifest, the generation will throw an error and the process will stop.

Name	Build
Description	The user wants to build the projects of the package in the workspace and get binaries ready to use on a given platform.
Trigger	The user. The validate use case.
Precondition	A workspace must exist and it should have its project files generated.
Flow	1. The user specifies a target toolset, platform and one or more architectures and configurations. 2. The system runs the required toolset to build the binaries for the package buildable projects.
Exceptions	• If the system fails to build any of the binaries, it will show an error and stop the process.

Name	Test
Description	The user wants to run the test applications to verify the status of the project.
Trigger	The user. The validate use case.
Precondition	A workspace must exist and its test projects must be built previously.
Flow	1. The user specifies a target toolset, platform and one or more architectures and configurations. 2. The system will try to run the test applications of the workspace package sequentially. 3. The system expects each application to return 0 on success and a different value to indicate a test failure.
Exceptions	• If some of the test binaries to run doesn't exist, the system will show an error and stop the process. • If some of the test binaries run returns a failure code, the system will show an error and stop the process.

3.5. Use cases

Name	Add project
Description	The user wants to create a new empty project in the workspace package.
Trigger	The user.
Precondition	A workspace must exist.
Flow	1. The user specifies a project name, location and other required attributes for the new project to create. 2. The system will add it to the manifest. 3. The system will create the required files and folders to make the package structure match the new manifest contents.
Exceptions	• If a project with the same name already exists on the package, nothing will be modified.

Name	Remove project
Description	The user wants to remove an existing project from the workspace package.
Trigger	The user.
Precondition	A workspace must exist.
Flow	1. The user specifies a project from the workspace package. 2. The system removes any references to it from the manifest. 3. The system runs the update use case to make sure the workspace stays in a consistent status after possibly having removed some dependencies.
Exceptions	• If the specified project doesn't exist on the workspace package, the system will show an error and the process will stop, leaving the manifest and the workspace as they were before. • Further exceptions can raise during the execution of the update use case.

Name	Add dependency
Description	The user wants to add a new dependency to a project from the workspace package.
Trigger	The user.
Precondition	A workspace must exist.
Flow	1. The user specifies which project in the workspace manifest to add the dependency to, and which dependency to add (name, version, where to find it...) 2. The system adds the dependency to the manifest as specified by the user. 3. The system runs the update use case, in order to check the dependency tree can be resolved and to download the new dependencies.
Exceptions	• If the specified parent project or the dependency don't can't be found, the system will show an error and the process will stop. • Further exceptions can raise during the execution of the update use case.

Name	Remove dependency
Description	The user wants to remove an existing dependency from a project of the workspace package.
Trigger	The user.
Precondition	A workspace must exist.
Flow	1. The user specifies which dependency from which project of the workspace manifest to remove. 2. The system removes the dependency from the manifest as specified by the user. 3. The system runs the update use case, in order to possibly resolve updated versions of the existing dependencies.
Exceptions	• If the specified project didn't contain the specified dependency, the system will show an error and the process will stop. • Further exceptions can raise during the execution of the update use case.

Name	Create branch
Description	The user wants to start a new branch of development, starting from the status of the workspace package.
Trigger	The user.
Precondition	A workspace must exist and its package must have VCS information.
Flow	1. The user specifies the name of the new branch to be created. 2. The system creates the new branch on the VCS starting from the current commit on the workspace package. 3. The system adjusts the package manifest to reflect the new location of the package.
Exceptions	None.

Name	Merge branch
Description	The user wants to merge the changes of a development branch into another one, of the workspace package.
Trigger	The user.
Precondition	A workspace must exist and its package must have VCS information.
Flow	1. The user specifies the name of the source branch. 2. The system will try to merge the changes from that branch into the current one. 3. The system adjusts the package manifest to make sure it's still in a consistent status. 4. The system runs the update use case to make sure it gets up to date with the possible changed dependencies in the source branch.
Exceptions	• If there are VCS conflicts when running the VCS merge, the system will show an error message and the process will stop. • Further exceptions can raise during the execution of the update use case.

3.5. Use cases

Name	Publish version
Description	The user wants to release a new stable version of the workspace package.
Trigger	The user.
Precondition	A workspace must exist and its package must have VCS information.
Flow	1. The user specifies the desired version number to release from the current status of the current branch. 2. The system runs the validate use case on the workspace package to make sure the current status of the branch is ready to be published. 3. The system modifies the workspace package manifest to reflect the changes needed for the release, like the version number, or any changes needed to lock the versions of the dependencies. 4. The system commits the available binaries, to make them available to the clients. 5. The system creates a VCS tag for the version.
Exceptions	• Further exceptions can raise during the execution of the validate use case if the package doesn't match some of the quality checks. • An error could happen if the system doesn't have write access to the VCS.

Name	Validate
Description	The user wants to check the status of a package.
Trigger	The user. The publish version use case.
Precondition	None.
Flow	1. If the user specifies a remote package information, the system will run the checkout use case with it. a. Alternatively, the system expects to find an existing workspace. 2. The system regenerates the workspace package project files by running the generate use case. 3. The system builds the workspace package binaries by running the build use case. 4. The system runs the workspace package test applications by running the test use case. 5. The system may run additional quality checks to validate that the package is ready to have a stable version released.
Exceptions	• The additional quality checks by the validate use case may show some errors and stop the process. • The execution of the checkout, generate, build and test use cases can raise their own errors.

4. Design

After the specifications analysis we're in a good position to start designing the system. The following sections will show the details and justifications of the design and the architecture.

4.1. Concepts

During the specification phase many new concepts appeared, and some of them need formalization. We need to define a common terminology to be used for internal reference but also when talking to the user. Here's a list of the most common related terminology.

- **Project:** It's a folder that contains a set of files that can be used together to achieve a goal. It may need other projects (dependencies) in order to work.
    - Some projects can be built, like applications or software libraries. In that case the project folder contains the source files.
    - Other types of projects can't be built, like header-only libraries or non-software projects, like data folders.
- **Project group:** It's a group of projects that are very closely related and that the developer will probably be working on at the same time, and the solution files will be created at this level. Examples of this could be a group of test projects or sample projects. This will be the smaller granularity that the system will be able to checkout separately, since the project files will require the whole project group to be available.
- **Package:** It's the main entity that the system can manage and it combines several related projects together. The objective of it is to offer a single public project that can be referenced as a dependency from the outside, and optionally a set of project groups used as support during the development or building of the main project. These support project groups are usually tests and samples, but there could also be tools or even libraries. The main criteria for moving a project to its own package is whether it's just being used for the development of this package or it could eventually be used directly from the outside.
- **Dependency:** It's a relationship between two projects where one needs the other to work. The typical meaning is that one project uses features of the other one, so both have to be linked together. Other than this, dependencies can also be used to specify that one project has to be built before the other one, specially in case of applications, where these helper applications can be used to generate code or pre-process some source files, for example.
- **Manifest:** It's a file included in a specific location of the package that contains the definition of the package contents and its relationships. It contains all the information that the system must know about the package:

- Package basic information like the name, version, branch, ...
- Project groups it contains, and which projects are inside of each.
- Build information like the supported platforms and architectures, and the build configurations. Premake file overrides can also be specified.
- The dependencies, both internal and external, between projects.
- The physical structure of the package, like which folders exist, and where the dependencies should be downloaded.

- **Workspace:** It's a folder in the user's hard drive that contains a main package, and it's where most of the actions of the system will get applied. It's considered by the system as the root of the user's work, and so it contains a special folder used by the system to identifying the workspace and to keep some temporary working files. The workspace can be a full checkout of the package, which will contain all of the project groups from the manifest, or a sparse one, which will only contain some of the project groups.
- **Shared folder:** It's a folder in the user's hard drive that's configured to contain the package checkouts that are shared among workspaces. It can be configured via an environment variable so each user can decide where it's best for him to store the shared packages.
- **Sibling workspaces:** These are workspaces which reside in the same parent folder as the main workspace. This represents a group of packages that the user wants to work on at the same time. It's common for simultaneous development of interacting packages. It can also be used to override the resolved version of a package with the version that the user has checked out on the parent folder. The dependency resolver first looks at the sibling workspaces for the requested packages before trying to resolve the specified version.

4.2. System architecture

In order to ease the distribution and installation of the tool, we decided to build a single standalone executable that would contain everything it needed to work. Since the application is written in large part using Lua, the Lua scripts would have to be embedded into the binary. In addition we also decided to distribute a version of Premake within the binary, so the tool was ready to be used without external dependencies.

The following is a descriptive list of the system components, as shown on Figure 12:

- GL-Get, our tool, consists of several components:
    - It's mainly developed using Lua, so the largest part of the system will be GL-Get's Lua scripts.
    - Where we needed some functionality that can't be implemented with Lua alone, we write C/C++ modules.

 - One of these modules is the scripts loader, which enables the Lua VM to load scripts embedded on the binary. The Lua VM itself knows how to load scripts from the file-system, but in this special case we had to build a bridge to give it access to the scripts that are only accessible from C code.
 - The last component is a Premake module that enhances Premake itself to make it aware of GL-Get and adapt it to our needs. This is only used when running Premake, during the generate command.

- Premake has a very similar structure (which we took as the inspiration for GL-Get): It's mainly developed using Lua, with some modules implemented using C/C++. It also has its own embedded script loader.
- The main application code is implemented using Lua, so we need a Lua virtual machine. Since Premake already provides a Lua VM of the most common version (5.1) we decided to reuse that one to run the main code, so that component is shared.

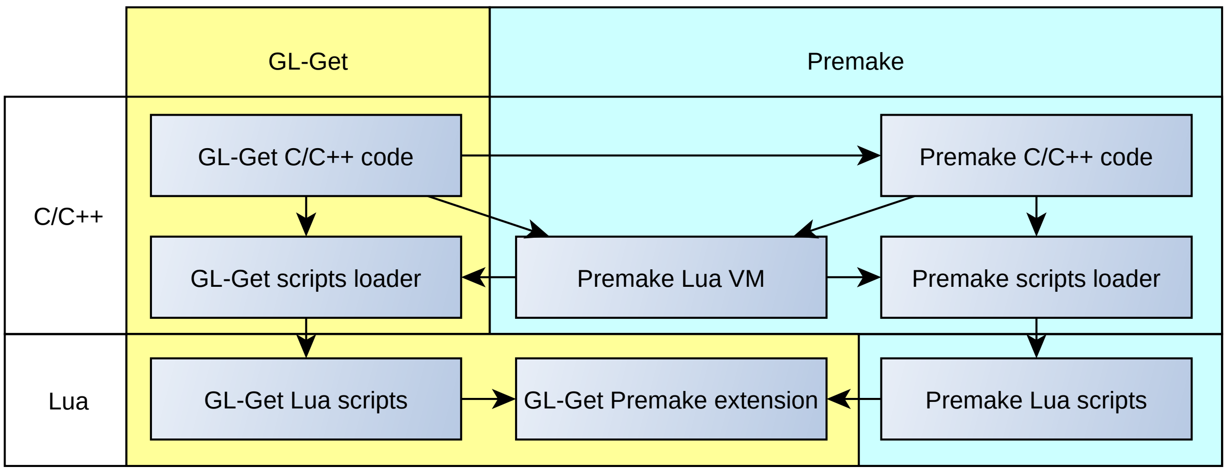


Figure 12: System architecture and components interaction

The system's entry point will be on the C/C++ code, which will instantiate and configure the Lua VM. The configuration part consists mainly on setting up the scripts loader so it knows how to load the embedded scripts, and also plug some custom C/C++ extensions so they're accessible from the Lua code (like, for example, to be able to call Premake from GL-Get's Lua scripts).

When the VM is configured, the C/C++ code loads and runs the main GL-Get script on the Lua VM and it takes control of the application. From that moment on, it can use the previously configured scripts loader to load additional Lua code. It has the ability to extract the GL-Get Premake extension code to the file-system, to make it available to Premake, and it also has the ability to call the embedded Premake. The rest of the application is implemented in the Lua code as will be detailed later on.

4.3. Application architecture

The whole system is designed to have a set of core modules, which will contain the common functionality, and a set of actions. The main application will take care of loading the configuration, setting up the environment, parsing the command line options, and it will delegate the execution to the requested action.

The following is a breakdown of the main parts of the system, going from lower level (general utility modules) to higher level (more specific to the application).

4.3.1. Actions and command line options

The application will be structured around actions, which run a given process. The main actions can be mapped almost directly to the use cases, but others can be added easily to automate processes making use of the core modules.

In order to automate and ease the process of specifying the command line options, we'll build a hierarchical structure (Figure 13) that organizes the description of the available options in contexts, so only the relevant options are shown and accepted:

- An option represents a single command line option that can be accepted. Each option can be configured with a short and a long names, a user visible description, an argument type, and optionally it can have a default value, it can accept multiple values and it can be made mandatory.
- An options group is a set of options with a title.
- An options context has a help message, a set of options groups and a set of sub-actions.
- An action has a name, a description, an associated runner function, and an options context.

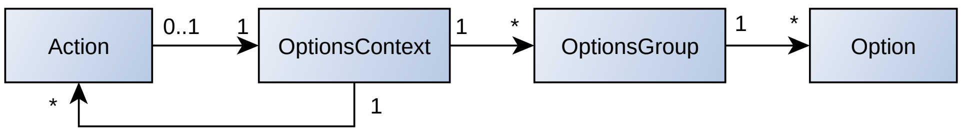


Figure 13: Options classes

Once the in-memory structure is filled, we can use it in many ways, like showing automated self-documenting help messages for the user (which will be available using the `--help` option on any context), or parsing the actual command line arguments (its main purpose).

The main idea is that in a context, all the options of any of its option groups are valid in any order (the groups are just used to help the user understand the available options). The options of the context will be parsed until a sub-action is found, at which point a new context will be opened, and its options will be parsed independently. At the end, each executed action (or sub-action) will be given the set of its own arguments and the sub-action's (if any).

For instance, the result of parsing the following command line:

```
app_name --option1 arg1 -o arg2 action1 --option3 arg3 action2
```

would result in these contexts:

- The first context would contain `"--option1=arg1"`, `"-o=arg2"` and `"sub-action=action1"`
- The second context would just contain `"--option3=arg3"` and `"sub-action=action2"`
- The third context would be empty.

4.3.2. Configuration

Many aspects of the application will be configurable, and we need a fairly flexible configuration system that allows the user to setup some global options and to allow overrides by using several configuration files. A simple example of something that everyone will have to configure is the VCS credentials.

Lua is often used as a configuration language too, which fits perfectly since we're already using it as the main programming language. The idea is that each configuration file will be a Lua script that returns a table that defines the desired options. Each configuration option will have a name (a string which will be used to access it) and an associated value. The value can be another table following the same ideas as the main one, allowing it to have a tree structure.

We want to allow the user to use several configuration files in order to progressively override the options specified in the most general configuration files. The configuration loader will run the configuration script, it will get the resulting table, and for each option of the override table it will try to merge it into the existing configuration:

- If the existing table doesn't contain that option, it will be set to the value from the override configuration (add a new option).
- If both tables contain that option:
    - If the values aren't tables, the old one will simply be replaced with the override one (replace an old value).
    - If the values are tables, the options from the override sub-table will be used to override the values from the existing sub-table, using the algorithm recursively.
    - As a special case, if both tables contain array elements, the elements from the override table will be added to the array of existing elements (concatenation of arrays).

In order to ease the initial configuration, we'll provide an embedded configuration file with sensible defaults. This initial configuration table will be the one used to merge the options of the more specific configuration files. From more general to more specific, the system will try to locate configuration files in the following locations:

- In the application’s embedded scripts (it will provide the defaults).
- In the home folder of the user (it should contain user’s global overrides, like VCS credentials).
- In the folder of the binary (it could contain version specific overrides, in case more than one version is installed).
- In the Workspace's temporary directory (it should contain just workspace specific options).
- In the current working directory (it should contain just local / temporary options).

Some classes can wrap particular configuration trees in order to add common query functions to them. For instance, the root of the configuration tree will be wrapped in a class that will provide, among others, a function that will return the path of the manifest file based on the manifest configuration. Other sub-trees that will be wrapped in a class are the VCS settings, which can be used to directly instantiate VCS connections or match location patterns. We’ll make use of this to allow the user to specify credentials for repositories that match a location pattern.

4.3.3. Logging

The logging facility allows the application to register the interesting events, which may be helpful for the user to understand what went wrong, and to help the developers to debug unexpected behaviors.

The current application doesn’t have very complex needs for logging, so we’ll go for a simple design:

- We define several log levels, which will represent different severity for each message. Each level can also have a color associated, which will be used when a message of that level is shown to the user. It will help the user identify the important messages (for instance, error messages will be shown in red). Here’s the list of available levels, sorted from more to less severe:
    - FATAL: An internal application error.
    - ERROR: Something went wrong, can’t proceed.
    - WARNING: Something may be wrong and should be fixed or needs the user’s attention, but we can proceed, keep an eye open.
    - INFO: General user information.
    - DEBUG: Extra information useful when debugging the application.
    - TRACE: Indication of an internal step which may be helpful to understand how an algorithm runs.

- The logger can be configured with:
  - A filename, where all of the messages from all the logging levels will be written.
  - The minimal log level that will be shown on the standard output. By default, the logger will show all messages of the INFO level or above (INFO, WARNING, ERROR and FATAL), which is most useful for the common user. In case the user, or a developer, wants to get more insight on why the application is behaving as it is, he can lower the logger level to DEBUG or even TRACE, which would show all of the messages.
- The logger will be a singleton with several methods, one for each log level, which will take care of processing the given message. When processing a message:
  - It's written to the log file.
  - If the current message's level is above or equal to the minimal configured one, it will be colored with that level's color, and it will be shown on screen.

4.3.4. Platform specification

In many situations, in the system, we'll need to specify a platform, an architecture, and other related elements. For instance, the generate, build, test and validate actions receive options that specify which toolset has to be used or which platform are we targeting. In these actions, each combination of these results in a "Job", which can be run independent from the other ones.

The main problem with this is that not all possible combinations are valid or make sense, so we need a way to select the valid ones. Once we have this information, we could also use it to deduce some of the parameters based on the specified ones.

First of all we need to define the main concepts:

- Platform: The target operating system, optionally combined with the run-time libraries (e.g. Windows, MacOS X, Linux, iOS, Android, Windows Phone, ...)
- Architecture: The target CPU / hardware instruction set (e.g. ARM, ARM64, MIPS, PowerPC, X86, X86_64, ...)
- Compiler: The toolchain that transforms the source code into binary form, which usually defines a unique file format that limits which objects can be linked together (e.g. Visual C++, GCC, Clang, ...)
- Build system: The tool that automates the build by using the compiler and linker on the appropriate files (e.g. gmake, MSBuild, Nmake, Xcode, ...)
- Toolset: A combination of the build system and the compiler (e.g. gmakeGCC = gmake + GCC, gmakeMinGW = gmake + MinGW, Visual Studio 2013 = MSBuild + Visual C++ 120, Xcode = Xcode + clang, ...)

4.3. Application architecture

As an example, table 4 shows several possible valid combinations.

Platform	Architecture	Compiler	Build system	Toolset
Windows	• X86 • X86_64	• VC*	• MSBuild	• VS*
			• NMake	• NMakeVC*
		• MinGW	• gmake	• gmakeMinGW
iOS	• ARM • ARM64 • X86 (Simulator)	• GCC	• Xcode	• XcodeGCC
		• Clang		• XcodeClang
MacOS X	• X86 • X86_86	• GCC • Clang	• Xcode • gmake	• XcodeGCC • XcodeClang • gmakeGCC • ...
Linux	• X86_64 • ...	• GCC	• gmake	• gmakeGCC
		• Clang		• gmakeClang
Windows Phone	• ARM	• VC*	• MSBuild • NMake	• VS* • NMakeVC*
Android	• ARM • ARM64 • X86 • MIPS	• GCC	• gmake • Ant • VS2015 • ...	• ...

Table 4: Example of valid combinations of platforms, architectures, compilers, etc.

The diagram in Figure 14 shows the chosen representation for this structure. In summary:

- An architecture is identified by its name.
- A platform is identified by its name.
    - A platform is available on one or more architectures.
    - On a given platform, each architecture could have an alias, used by Premake. For instance, the ARM and ARM64 architectures on iOS have the alias “Universal”, since it’s how Premake references them.
    - Optionally we’ll associate a value to the platform to let the system know which platform is the application actually running on.
    - Also optionally, a platform can contain a Premake option name and its associated value, which will be used to specify the platform to Premake.
- A compiler is identified by its name, and it can target several architectures.
    - When targeting a given platform, a compiler can only target a subset of its architectures, which will also be a subset of the architectures where the platform is available.

- A build system is identified by its name and the platform it targets.
  - A build system can use one or more compilers, which must be available for the platform the build system targets.
  - Additionally, the build system can specify if it can work with more than one architecture at the same time or not. If the build system can't handle it, the application will have to create several jobs to handle each of the architectures independently.
- A toolset is identified by its name and the platform it targets.
  - A toolset is a unique group of a build system and a compiler.
  - Optionally, an alias can be assigned, that would be used when talking to Premake (if it's different than its name).

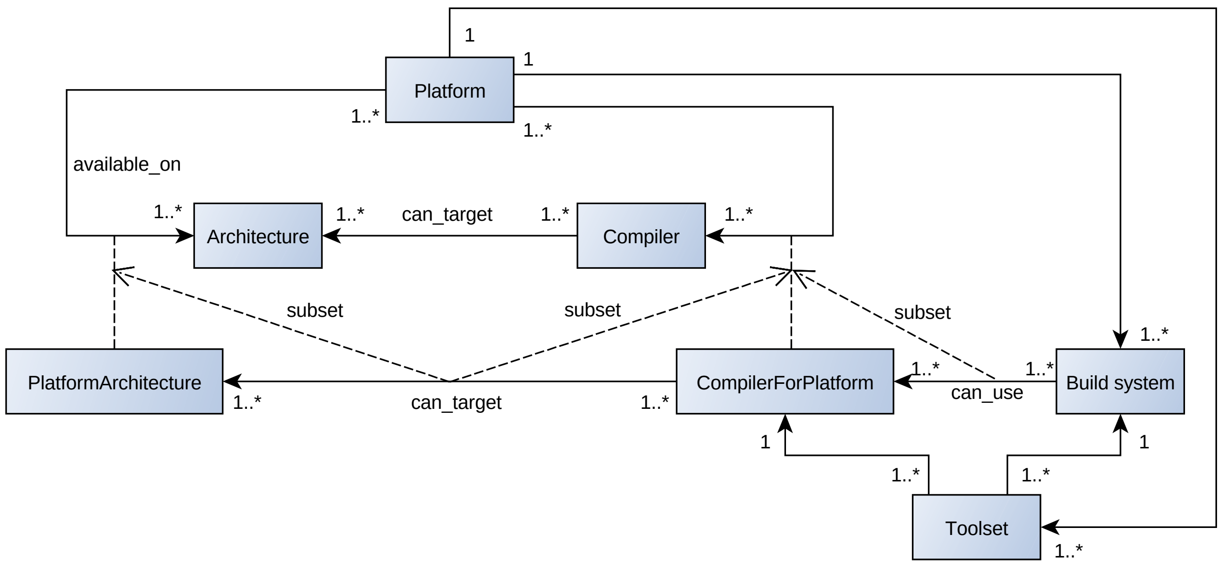


Figure 14: Valid toolsets definition

At startup, the system will fill the structures with the valid (known) platforms and their related information. This would finally allow the system to check if the given combination is valid or not.

Having all this information we can build a small deduction system that can give the possible values for an element given other ones. A few examples:

- From a platform we can access directly the available architectures, compilers, build systems and toolsets. We can easily extend it to get a list of multiple platforms and return a union of the elements of all the platforms (e.g. which build systems supported Windows or Android?)
- When given the name of one of the elements, we can iterate all the platforms to find the platforms that contain that element: for example, we can get which platforms can be built using Gmake.

Combining and extending these rules for the other elements, and with simple set intersections we can really improve the user experience. For instance, when running the application on Linux, the platform could be detected automatically, the set of available build systems would have a single element, which would be selected automatically, and the user would only have to specify the compiler or the toolset, depending on the intended action.

4.3.5. VCS abstraction

The code of the projects is stored in Version Control Systems (VCS), so we need modules to access them. Currently the vast majority of the company projects are stored in Subversion, but some of the last new projects have already been hosted using Git. Additionally, it would be very useful to have a mock back-end based on the file-system which we could use to run the test suite without depending on an actual VCS server running.

For this we'll have to create an abstraction to allow the system to work without having to know the back-end's details. The design is shown on Figure 15, and is based on the following principles:

- A VCSRepository represents the root of the remote server where the project history is stored.
    - It contains the common operations we need from a remote server, which are implemented in a subclass for each VCS type:
        - Creating a repository. It will be useful when creating a GL-Get package, but it may not be supported by all the VCS types.
        - Checking if the available information represents the root of a repository.
        - Listing the available branches and tags.
        - Doing a checkout of a specific branch or tag into a local working copy. This operation returns a VCSWorkingCopy.
    - A SvnStructuredRepository is added in order to factor out the similarities between the SVN and the FS mock repositories, which use a common directory layout to represent branches and tags. The following apply:
        - The root of a repository can only contain the trunk, tags and branches sub-directories. Trunk is the main branch, and the sub-directories of tags and branches are the names of the respective.
        - Creating a new repository could mean simply creating a sub-directory in a location which isn't a repository root.
- A VCSWorkingCopy represents a local checkout of a VCSRepository.
    - It has an association to the VCSRepository it was checked out from.
    - It can be created either by specifying the local path or when checking out from a VCSRepository.

- It contains the common operations needed to manage the working copy, which are implemented on a subclass for each VCS type:
  - Checking if the given directory is a checkout of the specified repository.
  - Updating to the latest commit of the checked out branch.
  - Committing the local changes to the repository.
  - Switching to a different branch or tag.
  - Setting the ignored filename patterns.
  - Managing sparse checkouts:
    - Listing the paths currently added to the sparse checkout.
    - Adding or removing directories to the list of repository directories we want on the working copy.
  - Managing the externals/sub-modules (linking a sub-directory to a different repository)

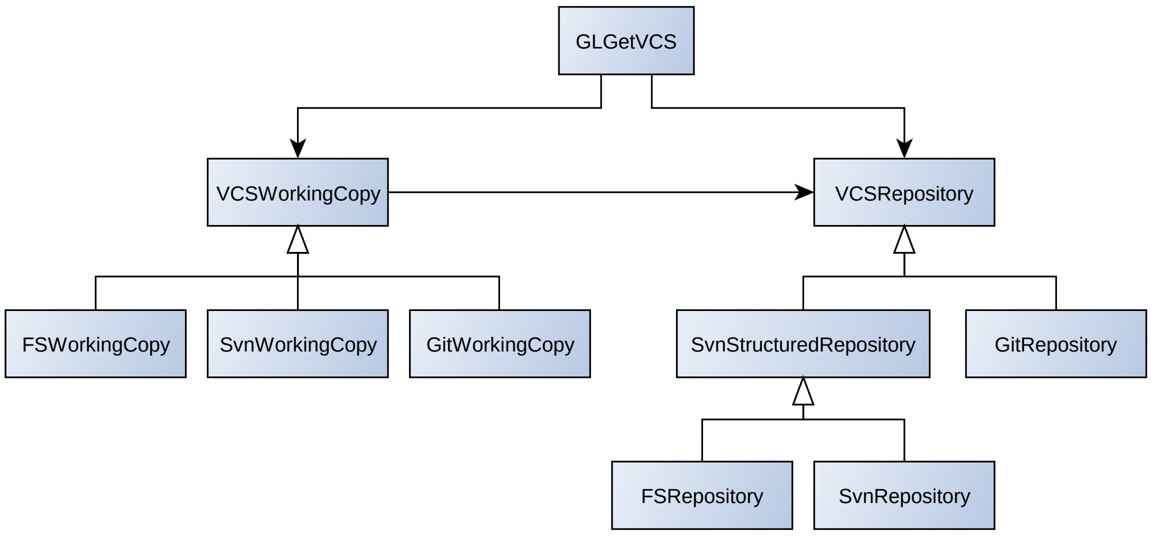


Figure 15: VCS abstraction

In addition to these general abstractions, we'll create a GLGetVCS class. It will keep track of a concrete working copy and a repository and it will contain a set of helper functions tailored for the application, like getting just the manifest of the package, or synchronizing a package, by checking the status of the working copy and doing a checkout or an update depending on whether a checkout was already in place or not.

4.3.6. Embedded Premake integration

One of the requirements of the system is that it should integrate well with Premake. Packages developed at different moments in time may need to use different versions, and different versions may be potentially incompatible. In order to ease the integration and deployment of the application we'll embed a version of Premake into the application, but at the same time we have to allow it to run external versions that can be downloaded as package dependencies. For this we'll create a very straightforward module which takes care of running the embedded version or an external one depending on the requested version number.

The application will have a way to query the version number of the embedded Premake. This could also be used for example to decide whether we have to download a Premake specified as a dependency or not. If the dependency targets the same version that's embedded, there's no need to download it.

Since Premake is a command line application, it's designed to parse its own command line arguments. The whole needed integration may be as easy as a single function that receives:

- The path where the external Premake is expected to be.
- The version of Premake that should be run.
- The command line arguments.

If the given version matches the one of the embedded Premake, that one will be run, with the given command line arguments. Otherwise, the external one will be executed, locating it on the specified path, and giving it the same command line arguments.

With this simple algorithm, the application will just "run Premake", and in the eventual case that the version matches, we'll automatically run the internal version (which will be a bit faster and won't use extra disk space). The implementation details are explained in section 5.3.2.

4.3.7. Embedded scripts

In order to make the application completely standalone, the Lua scripts (the source of most part of the application) would have to be embedded into the application:

- On one hand this can be thought as a simple resource container: we need to be able to get the contents of a file given its name.
- On the other hand we have to let the Lua VM know where to look for the required modules.

We can solve the first point by simply storing an array with the file names and another array with the corresponding file contents, both as static data in the executable. After a simple search on the file names array, we can get the file contents indexing the contents array using the same index.

Letting the Lua VM know about these scripts is a bit trickier, and will require implementing a Lua loader. This will be explained in detail in the implementation section 5.3.1.

4.3.8. Source extractor

As part of the system development we had to implement a Premake extension, described in section 4.3.14. Since we specified on section 4.3.6 that we'll be able to run arbitrary versions of Premake, we can't embed this extension with our own Premake, so we have to load it at run-time from the file-system.

The solution is to embed the Premake extension (which are plain Lua scripts) as part of the system scripts, but these aren't used from the main application. We create an extra Lua module that allows us to dump the files of the Premake extension into an arbitrary path of the file-system. After the scripts are written to disk, they can be opened by any version of Premake.

This module will contain a single function that receives a target path and will use the same data from the previous section to write the required files.

4.3.9. Build system

The system will have to use external toolsets mainly in order to run the compilation processes. Each of the toolsets has its own conventions, so we'll need a way to abstract these differences in order to decouple the system from the possible build systems.

We'll create a small class hierarchy, as shown on Figure 16, defining the common behaviors on the base class and leaving the implementation of the differences to the inherited classes.

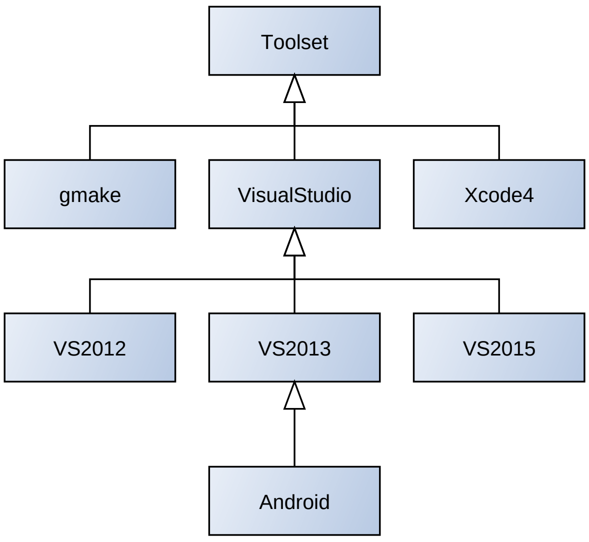


Figure 16: Build system abstraction

- The base class defines:
  - Methods for the building, cleaning and testing stages.
  - Additional pre- and post- methods for each of the stages, to be called before and after the main method respectively.
  - A function that will return the path where the project files are expected to reside.

- The implementation of the gmake toolset simply runs the "make" tool on the right folder and allows running the specified test by running it directly.
- Both the Xcode4 and the VisualStudio classes write build scripts that they later execute, in order to implement some commands that must be run together (like configuring the build environment). As an example, the pre_build method is where the script file is written, in the build method it's executed, and in the post_build method, the script is removed.
- In the case of VisualStudio, the pre_build step takes care of additionally creating a temporary solution file where the environment variables are manually replaced by their values. This is a known inconsistent behavior of the MSBuild tool, while the same file is processed correctly by the Visual Studio IDE (known bug at Microsoft, resolved as "Won't fix" [113]).
- Each of the VS* classes just contains the basic compiler and IDE version information to initialize the VisualStudio class properly (which are used to access the proper paths and environment variables).
- The Xcode4 class has some extra testing functionality to differentiate between MacOSX projects (which can be run natively on the system) and iOS projects, which need extra steps to deploy to a connected device and run.
- The Android module, in this special case, derives from VS2013 because at the moment we're using NVIDIA's CodeWorks for Android compiler [114], which acts as a Visual Studio extension, and so we can reuse most of VisualStudio's functionality.

The system is completed by a few helper methods that receive the job information (platform, architectures, configurations, path of the project files) and take care of instantiating the proper toolset class and executing all of the required processes.

4.3.10. Remote build system

Not all target platforms can be built from every host platform. In order to allow triggering builds for non-supported platforms, we'll develop a remote build mechanism. We'll rely on Jenkins as the remote build system and we'll create a simple Jenkins wrapper that allows us running the most common actions from our application.

This module will use Jenkins' public API, which uses JSON documents in order to transfer information. The main operations will be:

- Creating a job: it specifies the source repository and the procedure to run the build and test.
- Running a job: it will trigger a build.
- Deleting a job.
- Getting a list of a job's builds.
- Checking the status of a build.

With this we can easily add an option to the required commands to execute the build remotely.

4.3.11. Package definition

In order to identify the packages, we'll use a common base class that contains some shared behaviors. This will be useful when building manifests and dependencies, or even user requests from the command line, for example. First we'll list the information it will contain:

- The package name.
- The package version. This is optional, since we can't have it in all cases, like when dealing with legacy packages.
- The name of the repository where the package should be looked for. This isn't mandatory, but it will be necessary in case we're trying to locate the package and there's more than one repository configured in the application.
- The VCS location. This is an optional parameter, most useful when working with legacy packages.
- The VCS extra information, like the branch, tag, or revision. It's optional, and in case it isn't specified, it will be filled automatically with the available information (the tag corresponding to the specified version, the latest available version, or the main branch otherwise).

With all this information, the package definition can do many useful things:

- Tell whether it defines a legacy package or not.
- Check if the specified version and VCS information is valid, if it's missing information, or if it contains conflicting information (like in case both version and branch are specified at the same time).
- Build a descriptive string of the package it represents.
- Build the proper VCS setup to get the package.
- Check if the package exists on the VCS.
- Get the latest available version of the package.
- Build a valid folder name that identifies the package and its version.

All of these will be implemented as functions in the PackageDefinition class.

4.3.12. Manifest

The manifest is the main piece of information around which the whole application moves. Each package will be accompanied of a metadata file we called the manifest. This file contains both the package's own information (name, package version, internal structure, build instructions) and its dependencies' (which packages to get, where to get them from, which versions, where to store them...). This file will be stored in the VCS, so it's accessible at any moment by the application when we need to get a package or to resolve its dependencies.

We need a data structure to store the contents of the manifest in memory. The basic structure is shown on Figure 17 and is as follows:

- The manifest can contain several project groups. There must be at least one, which will be be the main one.
- Each project group can contain several projects, but they must have at least one. Project groups that contain a project identified as the main one, are restricted to that single project. Otherwise, project groups can have many projects. The main project group must have a single main project. That will be the only project that can be referenced from the outside.
- Both project groups and projects can contain dependencies, either directly or contained in folders. External dependencies (referencing another package) have to be specified inside a folder, just in case we ever need to do a local checkout of them. Internal dependencies, which are references to the same package, must be specified directly on the container, without any folder, because the same checkout will be used.
- Dependencies can target projects and project groups. Dependencies add the requirement that the target must be built before starting with the origin. If both ends are projects, and the target is a library, in addition the system will try to link that dependency with the origin project.
    - External dependencies can just specify the target package (either by repository, name and version or by URL). Package references resolve to the main project on the main project group of the target package.
    - Internal dependencies contain an identifier of the target project group and optionally the identifier of a project inside it. Internal dependencies can target either a project group (if only the project group is specified) or a project (if the project identifier is also specified).

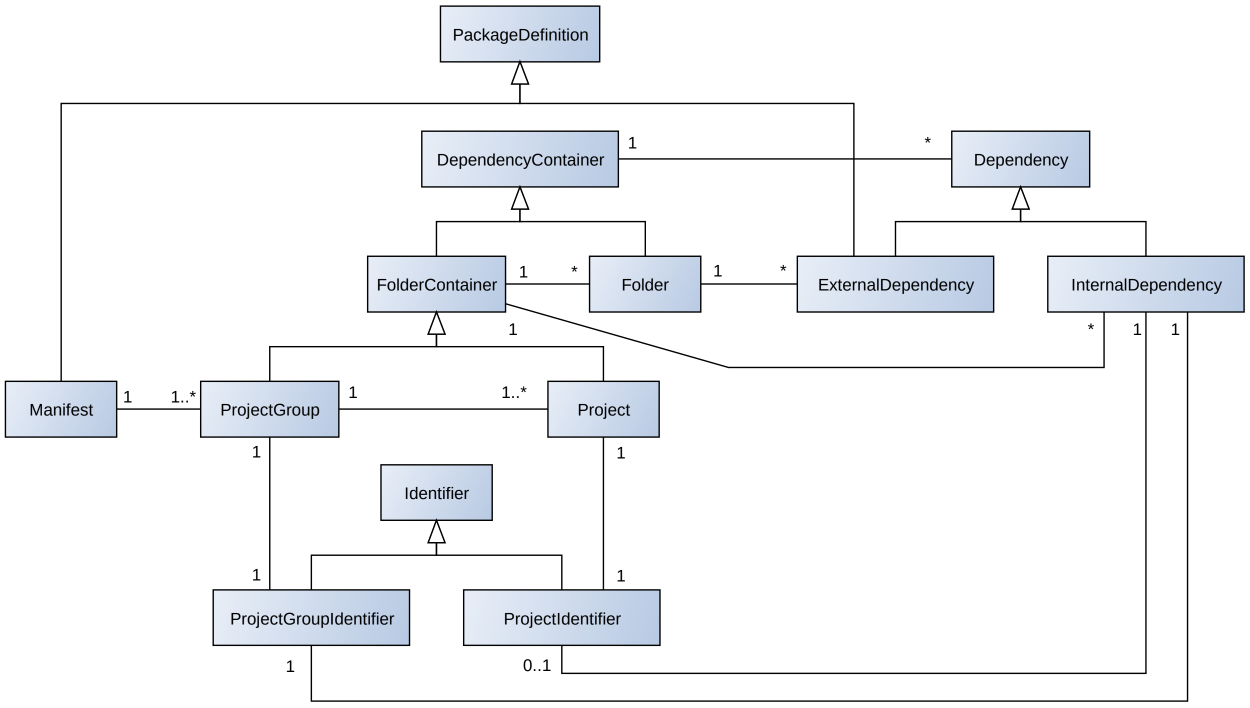


Figure 17: Manifest data structure

Other than the base structure, each of these classes can contain some extra information:

- The manifest contains the version of the manifest file specification used. Different parsers will be used based on this value.
- The package contains an identifier, like a version number or a branch or tag names.
- The project contains the project type, a UUID (needed by Premake to generate Visual Studio project files), an optional public Premake file, a list of configurations, a map of ABI flags for each configuration and the supported platforms and architectures.
- The dependencies can contain an option on whether they must be shared or not, a relationship flag that tells the kind of dependency (GL-Get, legacy, or Premake-enabled), an optional Premake file to set it up, a configuration map (which dependency configuration should be used for each parent configuration) and some profile and checkout mode mapping.

In addition to the in-memory representation, the manifests will be files on disk. We'll need a file format that we can parse. In order to follow Premake's conventions, the manifest files will be Lua scripts that can only call a well-defined set of functions. When called, these functions will take care of progressively building the in-memory representation of the manifest.

As an example, we show the manifest of GL-Get itself:

```
manifest_version "0.4.0"
package "GL-Get"
  version "0.0.6-dev"
  project_group {MAIN}
    folder "prj"
      dependency "GL-Get-Premake-utils"
        repo "libs"
        version "0.0.5"
    project {MAIN}
      type "app"
      uuid "C3A10768-5024-4119-BC75-346DF779DF8E"
      premake_file "glget.premake.lua"
      abi { "Debug:ds-er", "Release:os-er" }
      configurations { "Debug", "Release" }
      platforms {
        "Linux:X86_64",
        "MacOSX:X86_64",
        "Windows:X86,X86_64",
      }
      folder {LIBS}
        dependency "Boost"
          relationship "third-party"
          premake_file "Boost.premake.lua"
          vcs "svn"
          url "https://svn02/vc/libs/trunk/Boost"
```

Even if these don't look like it, each line represents a function call: when Lua functions take a single argument which is a string or a table, the parenthesis can be omitted. So the first line could also be read as `manifest_version("0.4.0")`, and the text indentation on the file is merely informative, it doesn't have any effect. When using constants, like *MAIN* and *LIBS* we surround them in brackets to convert it into a table so it can be used directly too. This way it doesn't resemble that much a programming language.

Since we have actions that can modify the manifest (like adding and removing dependencies, or creating projects), we'll need a manifest writer module. It will be trivial to write such module because it will just need to traverse the manifest nodes and write the relevant information as function calls that can later be interpreted by the parser.

4.3.13. Package cache

The package cache is a sub-system that allows storing shared information associated to a given package definition. Its main objective, which is built-in, is acting as a manifest cache, but other information can easily be stored.

The need for it is to reuse the already loaded and downloaded manifests: for example, if a dependency appears in several points during the dependency resolution, we just ask for the manifest of that dependency to the package cache every time we need it and it will only download it once. If a manifest was already downloaded on a previous session and we still have a local copy, it won't even be downloaded once again.

The cache is structured as shown on Figure 18:

- The PackageCache contains a list of PackageInfo objects, one for each package.
- Each PackageInfo represents a package without a specific version. It contains a list of PackageVersionInfo, which can be indexed by the package version string.
- A PackageVersionInfo represents a concrete version of a package.
- The PackageVersionCache is the actual object that will contain cached information about a version of a package.

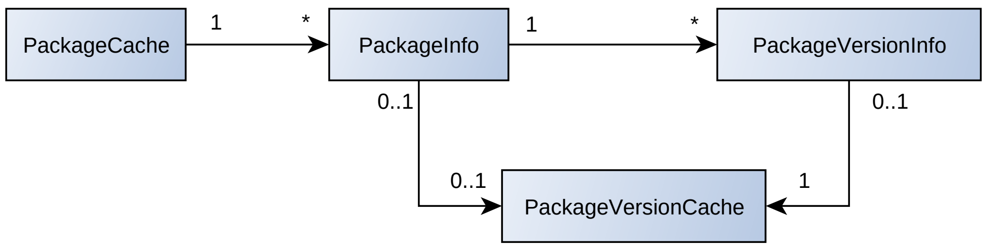


Figure 18: Package cache

The final objective is to access the proper PackageVersionCache, which can be located either in a PackageInfo (if we don't have any version information) or in a PackageVersionInfo. When a PackageDefinition is given to the PackageCache, it accesses the corresponding PackageInfo:

- If the PackageDefinition specifies proper version information, the corresponding PackageVersionInfo is accessed.
- If the PackageDefinition doesn't specify version information, it will try to download the manifest from the VCS configuration built by the PackageDefinition.
  - If that manifest contains version information, that will be used to access that version's PackageVersionInfo, and the manifest will be cached. At the same time, the resolved version will be stored on the PackageInfo so it can be used in case there's another request without version information.
  - If no manifest can be obtained, the PackageInfo object will be used as the final location.

Once we decide which is the proper parent object for the requested PackageDefinition, we access the PackageVersionInfo it contains, and create one in case it doesn't contain any. That's where we'll store all information we know about that specific package version, and we'll begin by storing the VCS configuration or the manifest if we have any of them. Otherwise, the PackageVersionInfo will know how to get them when asked for.

4.3.14. Premake extension and translation tree

Premake creates project files for many platforms, which is a very useful task, but it knows nothing about package dependencies, version numbers, platforms and so on, so it has to be given all the information it needs. GL-Get has all of this information, which is extracted from the packages' manifests, but we need a way to communicate this information to Premake. In order to implement this communication, we'll need two components: a data file where this information is written, and a Premake extension that reads it and knows how to handle it.

The data file is called translation_tree.lua for historical reasons, and it will be written next to each solution's premake5.lua file. It contains two main blocks of information:

- Toolset information that can be used to deduce the toolset, compiler, platform and architecture names from the action and the options passed to Premake. We need all this in order to build mangled filenames for the target folders and binaries. The information on this table will be pre-processed, so it's ready to use with the information available in Premake, instead of using the complete structure from GL-Get (described on section 4.3.4).
- Solution information, which contains the information needed by Premake, like its name, the supported platforms and architectures, the configurations, and the list of projects. For each project we'll contain many things, like its name, project type, paths, location of the Premake snippet, the ABIs map, configurations, the supported platforms, and a list of its dependencies. Each dependency will specify its name, version, project type, relative path, type of package, and the maps of ABIs and configurations, with the ones corresponding to each element from the container project.
- Optionally, in case it was included as a project group dependency, it contains the version of the Premake-utils package, so it can be loaded automatically.

In order to use this information, we'll write a Premake extension that adds new features to the existing commands. Its main objectives will be:

- Reading the translation tree file and adding helper functions to easily query information from it.
- Automatically loading the Premake-utils package in case it was a project group dependency. It will be located in a sub-folder of the project group, and its folder name will contain the package version, so we need this information from the translation tree.
- Adding some functions that do some work automatically based on the information from the translation tree. For example, we'll have a function that sets the proper target name and target path for a specific configuration of a project. Another example would be a function that automatically adds a dependency to a project by just giving it the dependency name: it would use all the information available about that dependency, and call the proper Premake functions to set it up.

- Extending Premake's commands so they do some work automatically. For example, the *solution* function would take care of setting up the solution's architectures and configurations automatically. The *project* function will just take a project name and it will automatically setup everything it knows about it: its path, the configurations, its target names and the dependencies.
- Modifying some of Premake's commands so they take into account the package they're working on. For instance, when the *includedirs* function is called with a relative path, it will be applied to the base path of the active package (like a dependency).
- Writing a file that contains the final filenames and paths after being processed by Premake. This step is needed in case the user adds some customization to the solution's Premake file, and for some complex targets that may use non-standard paths. This information is later read back by GL-Get so it knows how to run the tests.

With this we close the Premake interaction both ways, making it fully aware of the packages' information, and making GL-Get aware of what Premake used to build the project files.

4.3.15. Package resolution

Starting from the main loaded package, we want to know what versions of which packages are needed in order to make it work. For that we'll build a resolution tree, as shown on Figure 19. It can be thought as a structure parallel to the manifest, which contains augmented information about the concrete instance we're looking at. The same package (manifest) can be referenced in different points of the tree, but not all instances of the resolution sub-tree may be equal.

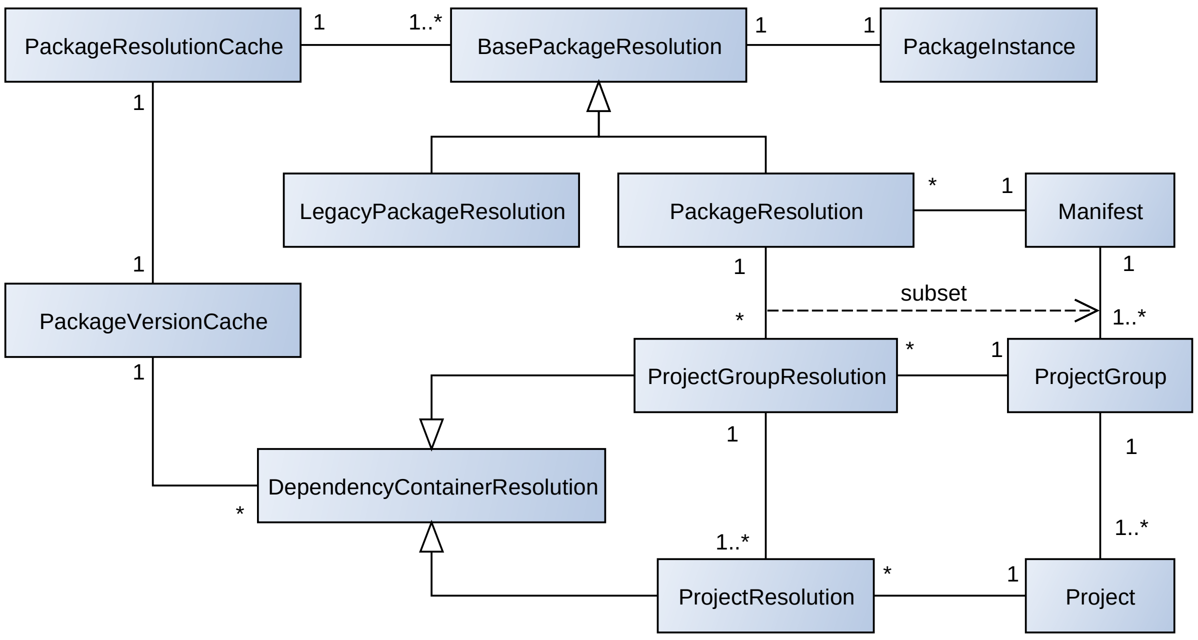


Figure 19: Package resolution tree

The PackageResolution keeps track of the set of enabled project groups of that concrete instance, which can be different based on the requested profile or enabled arbitrarily by the request from the outside.

The interesting part is how the PackageResolution of the dependencies are referenced from the DependencyContainerResolution:

- We add a PackageResolutionCache to each PackageVersionCache (which was described in section 4.3.13) that's used for the dependency resolution. Its objective is to keep track of all the different PackageResolution trees that can be created for the same package version. With this, when a package appears more than once in the dependency tree, that specific sub-tree can be reused.
- The DependencyContainerResolution doesn't keep the references to the dependencies' PackageResolution, but it uses the PackageVersionCache to retrieve them whenever they're needed. With this we automatically can get the latest version resolved for that scope.
- While creating the PackageResolution, a structure will be created dynamically to represent the link scopes of the applications, which will be passed to their library dependency projects. The link scope is where the conflict resolution is done when two instances of the same package are found:
    - If both instances reference the same version, everything's fine and the process continues.
    - If both instances reference different versions:
        - If they're compatible (as defined by Semantic Versioning [115]), the newer version is chosen, and it will be used for any instance requested inside the link scope.
        - Otherwise, it's considered a conflict and the process will throw an error.

The package resolution tree is a central part of the system, and it's used for many things. The only algorithm embedded on the structure is the loading itself, but the rest are written separately. In this case there's a single algorithm, which builds the structure described on the next section. Many other algorithms run from that structure but also access the package resolution tree.

4.3.16. Package checkout

After doing the package resolution, it's convenient to have an easily accessible list of packages and their associated checkout information (PackageCheckout), which would be irrelevant on the resolution tree. In some cases, several nodes on the resolution tree may reference the same package checkout on the hard drive (like in the case of packages that should be checked out to the shared folder), so this package checkout information is a flattened list that has all these duplicates merged. This allows to easily know which packages have to be checked out and to track the status of their synchronization to disk.

First of all we'll model the checkout location (Figure 20). It has two purposes:

- Giving access to the list of packages that have to be checked out to a specific location. In the case of the workspace checkout location, it will only contain a single package, but all other locations can contain more than one.
- Abstracting the concrete path for a given package. For instance, when asking for the package path to the workspace checkout location, it will return the root of the workspace, but all container checkout locations will return a sub-directory of them. The directory naming convention will be different for each container:
  - in the workspace siblings, each package will only have the package name on the directory name
  - on the shared folders it will contain the package name and a combination of the VCS information (package version, branch, revision, etc)

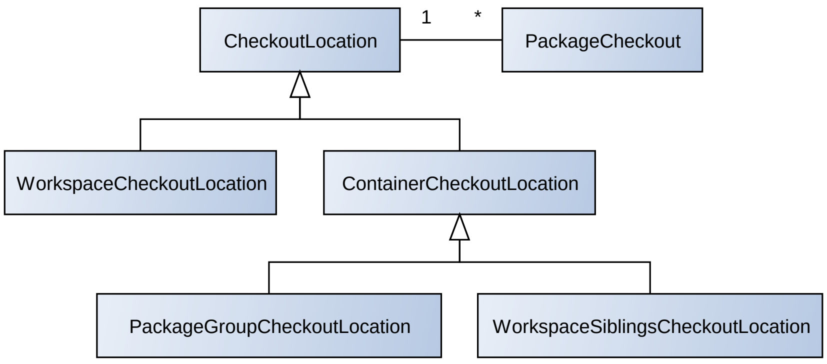

CheckoutLocation
1
*
PackageCheckout
WorkspaceCheckoutLocation
ContainerCheckoutLocation
PackageGroupCheckoutLocation
WorkspaceSiblingsCheckoutLocation

Figure 20: Checkout location

For each package, we can get requests for either:

- Some of its project groups, and these will be associated to one or more checkout modes. The checkout modes are "source" and "binary", which indicates whether the source folder of the project group will be checked out, or the target one, or both. We'll create the ProjectGroupSelection class to model this (Figure 21).
- A profile, which will map to a predefined ProjectGroupSelection. There are several defined profiles, like full (all project groups with all checkout modes), main (for just the main project group with a specified checkout mode, the default for the dependencies), and devel (for the main and the tests project groups, the default for the main workspace package).

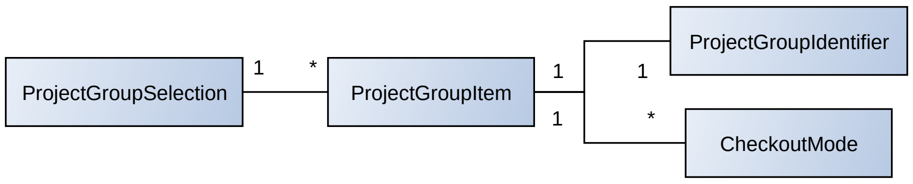


Figure 21: Project group selection

Given the algorithm difference when doing a full checkout or a sparse checkout, we'll draw on the strategy design pattern, and create the checkout strategy (Figure 22). It encapsulates the algorithm that knows how to checkout or update a package, given its location on the hard drive, its VCS information, and the list of the requested project groups and their checkout modes (the ProjectGroupSelection).

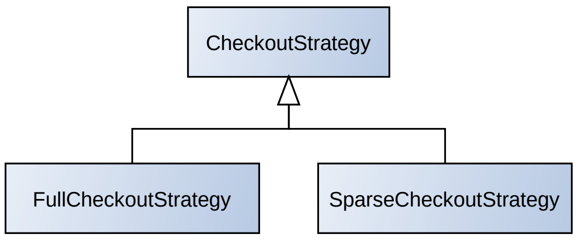


Figure 22: Checkout strategy

Finally, the package checkout (Figure 23) will be the one containing all this information:

- It contains an association to the PackageInstance from the dependency tree that contains the whole context from the dependency resolution process.
- It has a CheckoutStrategy that tells how this package should be checked out.
- It has a ProjectGroupSelection that tells which project groups should be checked out and their requested checkout modes.
- It has a CheckoutLocation that helps it compute its target path. In addition, one could access the PackageCheckout from the CheckoutLocation, which is a good way to trigger the synchronization process.

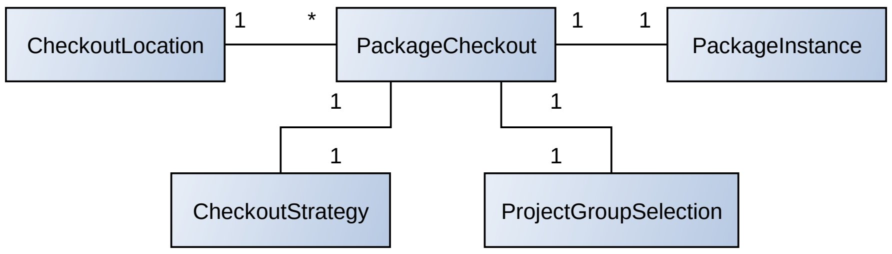


Figure 23: Package checkout

Similar to the package resolution tree, the algorithms are also separated from the package checkout information:

- Sync: synchronizes the structure to disk. It navigates the structure by creating the needed directories and checking out or updating the packages as needed.
- Write translation tree: it writes to disk the translation tree file described in section 4.3.14 for each enabled project group of each described package. For it, it accesses the package resolution tree, where it can find all the dependencies information.
- Generate: it runs Premake on each enabled project group of each described package.
- Build: it builds the projects of every enabled project group of each described package.
- Test: it runs all the executable tests built from every enabled project group of each described package.

4.3.17. Workspace

The workspace has two main responsibilities:

- It represents the physical directory where the main package is located and it handles everything related to its files (managing of its temporary sub-directory, manifest cache, local configuration, extraction of the Premake extension into the temporary sub-directory, etc). When loaded properly, it contains a reference to its manifest and it takes care of triggering the dependency resolution.
- On the other hand, it acts as the system facade with which actions interact most of the time. It contains the functions that do most of the work for the use cases: update, generate, build, test, add project, add dependency, etc.

Having this facility, implementing a new action is as easy as handling the command line arguments, loading the workspace, and delegating the work to the desired workspace functions.

5. Implementation

This chapter explains the most relevant implementation details, like the generally used techniques or libraries. The implementation of the particular modules is often straightforward, given the design from chapter 4, so it won't be detailed in this chapter.

5.1. Lua code

Lua is a very nonrestrictive language, which allows the developer to implement things in many different ways. For instance, the Lua-users wiki [116] contains lots of pages discussing possible solutions to many common subjects on Lua, from common solutions to proposals for standardization or useful tricks. We had to agree on our own conventions, otherwise it would've been hard to make the several modules to integrate correctly.

5.1.1. Arguments checking

Since Lua is a dynamically typed language, it would be a good idea to add some kind of checking to the functions we wrote. For that, we agreed that we would add asserts to check the type of the arguments at the top of the functions, with all the known restrictions at the time of writing the code. This is an example:

```
local function f(arg1, arg2, arg3)
  assert(type(arg1) == "string")
  assert(type(arg2) == "table" or arg2 == nil)
  assert(SomeClass:class_of(arg3))
  -- Do the work...
end
```

5.1.2. Modules

Lua has a function called *require* that allows loading other modules. When requiring a Lua module, the interpreter uses a list of patterns to expand a file name from the module name, and it tries to open them until an existing one is found. Once a file is found, that module's code gets executed, giving complete freedom to the module developer on what he wants to do at that point.

- Older modules just set global variables that can later be accessed from the caller module (and any other module), but it results in global namespace pollution, which could lead to name clashes, which would just result in the last module with the same name overwriting the previous ones. Also, the module is the one that decides the name under which it would be visible. Working directly on global variables opens the door to undesired side-effects. The following snippet shows how a module would be implemented using this convention:

```lua
my_module = {}

function my_module.func1(a, b)
  return a + b
end

function global_func(a, b)
  return a - b
end
```

That module could be used like this:

```lua
require "my_module"

local a = my_module.func1(1, 2)
local b = global_func(2, 1)
```

- The newer convention for module implementation is to define everything in local variables and offer the relevant functionality in a table returned at the end of the module execution. It has the advantage that the global environment isn't modified by default, giving the choice to the user. In addition, the module could be used through an alias, helping avoid name clashes when there's lots of modules. The following is an example of a module definition using this convention:

```lua
local my_module = {}

function my_module.func1(a, b)
  return a + b
end

return my_module
```

In order to use that module, the user could use it like this:

```lua
local that_module = require "my_module"

local a = that_module.func1(1, 2)
```

In the end we decided to go for the latter, given its advantages on name scoping.

5.1.3. Classes

Lua was designed as a procedural language with no representation for classes. Fortunately, its tables are generic enough to create some basic object orientation:

```
local obj = {
  value = 1
}

function obj.func(object, arg1)
  return object.value + arg1
end

a = obj.func(obj, 1)
```

On this simple example we wouldn't need the *object* argument, since we're always working on the same table, but we keep it to match the following examples. The main problem of this approach is that both data and functions reside on the same table.

To circumvent this limitation, Lua provides metatables. Each table can have a metatable associated, which can be used to define some of the table's behaviors:

```
local t = {} -- This is a simple table
local mt = {} -- This is another simple table
setmetatable(t, mt) -- Now mt is the metatable of t
```

The simplest behavior that can be specified is what would happen when a non-existing field of a table is requested. This can be specified using the metatable's *__index* field. If a function is assigned to that field, it would be called to get the values of non-existing fields of the table. Alternatively, if a table is set on the *__index* field, the requested field would be looked for in that table.

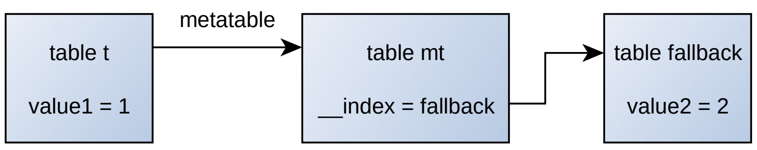


Figure 24: Fallback values using metatables

The following example shows how this fallback method would work in practice (Figure 24 shows the relationships between the tables):

```lua
local t = { value1 = 1 }
print(t.value1) -- This would print "1"
print(t.value2) -- This would print "nil"

local fallback = { value2 = 2 }
local mt = { __index = fallback }
setmetatable(t, mt)

print(t.value1) -- This would print "1"
print(t.value2) -- This would print "2"
```

With this we could create tables with arbitrary fields (which could contain functions) shared among several other tables. This is used commonly in Lua to implement prototype-based programming, a style of object-oriented programming popularized by JavaScript.

In addition, some syntactic sugar was added to help writing object oriented code. The colon character can be used instead of the dot to represent member methods. When declaring a function with the colon character, it adds a new implicit parameter called *self*, which represents the object where the method is being called, similar to the *this* pointer in C++. When calling a table function with the colon character, it implicitly adds the table as the first argument to the function. Here's an adaptation of the previous example using a metatable and the colon syntax:

```lua
local MyClass = {}
local MyClass_mt = { __index = MyClass }

-- This is equivalent to: function MyClass.func(self, arg1)
function MyClass:func(arg1)
  return self.value + arg1
end

local obj = {
  value = 1
}
setmetatable(obj, MyClass_mt)

-- This is equivalent to: a = obj.func(obj, 1)
local a = obj:func(1)
```

There are many tricks and conventions through the Lua ecosystem around object orientation, leading to many different implementation details. Since we were used to class-based object orientation, we decided to use Penlight's [117] implementation for the class system. That offers some extra functionality like inheritance, constructor chaining and derived type checking. The following is a small example of how it can be used:

```
local class = require "pl.class"

local ParentClass = class() -- Creates a new class
function ParentClass:_init(val) -- Defines the constructor
  self.val = val
end
function ParentClass:add(val) -- Defines a method
  return self.val + val
end

local DerivedClass = class(ParentClass) -- Creates a derived class
function DerivedClass:_init(val)
  self:super(val + 1) -- Calls the parent class' constructor
end

local obj = DerivedClass(2) -- Creates an instance
local a = obj:add(1) -- Returns 4
assert(DerivedClass:class_of(obj)) -- obj is a DerivedClass
assert(ParentClass:class_of(obj)) -- obj is also a ParentClass
```

5.1.4. Error handling

There are many conventions in Lua to handle errors:

- The native Lua errors are generated using either the *error* or *assert* functions. Run-time errors, like trying to add “1 + *nil*” also raise this kind of error. These behave in a similar way to C++'s *throw*, finishing the program execution if they're not captured. The way to capture these errors is by running the code that throws the error in a protected environment, by wrapping it in a function called using a protected call (using the *pcall* function). In case of an error, *pcall* returns *false* plus an error message. Otherwise it returns true and all the values returned by the called function. Here's an example of *pcall* usage:

```
-- Original call:
local result = func_with_error(arg1, arg2)

-- Protected call:
local status, result = pcall(func_with_error, arg1, arg2)
if status then
  print("OK!")
else
  print("Got an error:", result)
end
```

The main disadvantage of this method is that the call code is much less readable than the original, specially when dealing with classes, as it can be seen on the following example:

```lua
local obj = MyClass()

-- Original call:
local result = obj:func_with_error(arg1, arg2)

-- Protected call:
local status, result = pcall(obj.func_with_error, obj, arg1, arg2)
-- handle the status value...
```

- Another option is to design the functions that can fail to return numerical error codes. That's similar to the way the error code is used on the shell scripts to determine if the execution of an application finished successfully or not. It's common to return 0 when everything went fine and another number in case of failure. An issue with this is that it's hard to know the meaning of each value from the outside, and their meaning could easily change when possible return values are added or removed.

- An option used by several functions in the standard Lua run-time and some external libraries is to make the return values compatible with the arguments expected by the *assert* function: the first argument is interpreted as a boolean, with values of *false* or *nil* meaning that an error should be raised. In case of error, the second argument is used as the error message. Otherwise, the assert function returns all the arguments passed to it, acting as a bypass.

  The following example shows the implementation of a function following this convention:

```lua
local function f(a, b)
  if type(a) == "number" and type(b) == "number" then
    return a + b
  else
    return nil, "Both arguments must be numbers"
  end
end

local a = f(1, 2) -- Unchecked call
local b = assert(f(1, 2)) -- Asserted call

-- Checked call:
local c, err = f(1, 2)
if err then
  print("Got an error:", err)
end
```

  The main advantages of this approach are that it doesn't complicate the syntax of the code, the error checking is optional, and it can easily be wrapped in an assert call in case we want to promote the type of error.

While numerical error codes were used initially in the prototype implementation, we decided it didn't provide the flexibility and semantics we needed. So in the end we established these rules:

- Error and assert should be used to protect the code. These shouldn't happen and they point out bad code usages.
- Functions that could fail (have alternative code flows) depending on external data, like user input, or file data, should return assert-compatible values. These can easily be raised to errors wrapping the call with assert() in case it's needed.

5.2. C/C++ code

Most of the C/C++ code in the project is either very simple or is a direct translation from the designs on the previous chapter, and its implementation has no special interest. C++ supports all the features required by the designs above, like classes, so nothing special was required. The most interesting part may be the creation of native modules for Lua.

Lua is a very simple language which was designed to be easy to embed, so it's easy to make it interact with other languages. Its interface is implemented in C, and its main communication channel is the virtual machine's stack. The main types of interaction are:

- Pushing or popping values to/from the stack. Arbitrary C/C++ values (even memory pointers) can be stored on the Lua VM by wrapping them on the *userdata* structures, though these appear as an opaque data reference from the Lua language side, so it's better trying to avoid them. These can later be used for anything (function parameters, storing into a table field, etc).
- Calling a function: one has to push the function and its arguments to the stack, and the return values will be located at the top of the stack after returning.
- Creating a C function that can be called directly from Lua: these functions must have this prototype:

```
typedef int (*lua_CFunction) (lua_State *L);
```

  The lua_State is the current state of the virtual machine, which can be used to get the function arguments and to access the rest of the environment. The function must return the amount of returned values. Inside these functions, the implementation can use C++ code as complex as it wants.

The implementation of a native Lua module using C/C++ usually consists of the following steps:

1. Implementing the desired features using the most convenient language (often C++ classes).
2. Writing the C functions that will serve as the interface to the module, which will bridge to the code written on the first step.

3. Registering the interface functions as fields of a table, so all the functions can be accessed like module.function1, module.function2, etc. There's some helper functionality for creating modules by just providing its functions.

The final result from the point of view of the Lua language is a module that behaves like the ones described on section 5.1.2.

5.3. Standalone executable

As specified in the non-functional requirements, it was expected to distribute the tool as a standalone binary with its dependencies embedded to simplify its download, installation and configuration. This section explains how we packaged all the components shown in Figure 12 together, and how they're coordinated at run-time.

5.3.1. Embedded scripts

To be able to have a single executable that contained the system scripts, we had to embed the scripts as resources and implement an embedded scripts loader. The Premake extension scripts are used in a different way, but we embed them together with the system scripts to reuse the same embedding system. The easiest way to embed the scripts as resources was to generate a .cpp file which contained the scripts as C strings.

We used Premake's scripts embedder as a base to build our custom one. It's a simple Lua script that runs as a Premake action and it takes advantage of some of its features to get lists of files from a given pattern. As a result, it generates a .cpp file with the following structure:

```
const char* k_builtinScriptsIndex[] = {
  "filename1.lua",
  "filename2.lua",
  ...
};

const char* k_builtinScripts[] = {
  "contents of filename1.lua",
  "contents of filename2.lua",
  ...
};
```

With this simple structure, the embedded script loader can search the first array for the file name, to later get the corresponding file contents by indexing the second array with the same index. The embedded script loader is implemented as a C Lua module, as described in the 5.2 section.

As explained in the reference of the *require* function [118], a Lua script loader consists of two main parts, which we implement in C because they both need to access the scripts arrays:

- A module loader function: this is given the module name and it must actually load the module into the Lua VM and return its main value. We make use of Lua's own script loader functions to load the script from the in-memory array.
- A module searcher function: this is given the module name and it must decide whether it can load the module or not. In our case, this function simply checks whether the filename exists in the file names array. If affirmative, the function returns the embedded module loader function, which can later be used to load the specified module.

  The module searcher function has to be injected into the *package.loaders* array. This array controls the order in which different searchers are tried, so we'll have to insert our searcher after the file-system searcher.

Every time a module is required from Lua, the *require* function will first use the standard file-system module searcher, loading a module from an external file if it's found. Otherwise (what we expect for most of the time), it will fall back to our new embedded scripts searcher, which will look for the module name in the embedded scripts file names array. If the module is found, the embedded module loader function will be used to load it, which will parse the contents of the embedded file.

With this we implemented a simple override system that allows us to simply drop Lua modules in the file-system and these will be loaded instead of the embedded ones if their filename matches. This is very useful during the development process so we don't need to recompile the new embedded files every time we want to test a change.

5.3.2. Dependencies

The main system dependency is a Lua virtual machine in order to run its own scripts. Lua is developed as an embedded library, so it was just a matter of linking the application together with the Lua library. Other than that, the application requires Subversion and Premake to execute some of its commands, but these aren't vital to make it run.

Subversion is already installed in most developers' machines and it's properly configured in the PATH environment variable. In the future we plan to add support for Git, so Subversion could become non-essential. For these reasons we considered it's better not to to embed Subversion. Moreover, if necessary, the svn executable path can easily be set in GL-Get's configuration.

Premake is the last big dependency. Initially we relied on an external Premake which the user had to properly setup on its system. The way we've been using Premake in the company, there has been a copy on a sub-folder of each project, and each project could have a different version, so there wasn't a global installation or a proper reusable configuration. Since this is considered a basic tool for our current work-flow, we decided to embed Premake into GL-Get.

Since Premake already includes a Lua VM, we decided to reuse it, as shown in Figure 12. The Lua VM from Premake contained some small customizations, mainly to plug their own embedded scripts loader. We had to do a few modifications so it could be used for GL-Get before using the embedded Premake for the first time.

5.3. Standalone executable

Premake isn't developed as a library that can be embedded in another application, but as a main application itself. The main trick we had to apply was to rename Premake's *main* function so it didn't collide with GL-Get's:

```
#define main premake_main

namespace premake
{
extern "C"
{
#include "premake/src/host/premake_main.c"
}
}
```

Besides this, we also had to write a C Lua module (as described in the 5.2 section, following the design from section 4.3.6) that allowed us to run Premake from Lua. It basically takes all the function arguments and builds an array from it, which can be given to *premake_main* like it was its own command line arguments array.

5.3.3. Main application

The flow of the main application can be simplified as follows:

- A Lua VM is created.
- We do some basic processing of the command line arguments that can't be done at a later stage (like trying to get the name and absolute path of the executable), and package all the arguments into a Lua array so they can later be accessed from the scripts.
- Setup the C Lua modules:
    - A console module which overrides Lua's own *print* function to make it convert the ANSI terminal color codes into Windows API calls, to make the colored output to work on Windows.
    - The command line arguments parsing module, which was described in section 4.3.1.
    - The embedded script loader, described in sections 4.3.7 and 5.3.1.
    - The scripts extractor, which is used to dump the Premake extension scripts, described in section 4.3.8.
    - The LuaFileSystem [119] module.
    - The embedded Premake module described in section 4.3.6 and in the previous section.

- We set the *package.path* Lua variable, based on the command line arguments, to let it know where the scripts can be located on the file-system.
- Load the main GL-Get script using the *require* Lua function. It's aware of the embedded script loader, so the main script can already be loaded from the embedded collection.
- Finally, we delegate the rest of the execution to the main function of the main script. The value returned by the script is returned to the operating system.

5.4. Bootstrapping

One of the trickiest parts of the implementation is creating the first build of GL-Get. GL-Get is packaged as a GL-Get package itself, so we need GL-Get in order to generate its project files and to build it. This chicken-or-egg problem is common in complex computer software which needs itself to work, with compilers as a common example, since they're usually written using their target language.

The process used to get an initial working version of the software is called bootstrapping. The bootstrap process is different depending on the requirements and the actual implementation of the system, so we had to implement a bootstrap system for GL-Get.

We used Premake as a reference, which also uses itself to generate its project files. Premake has a special mode (with extra code to support it) which lets it work without the embedded scripts, loading them directly from the file-system in a controlled environment. Using that, a basic Premake version can be built with custom Makefiles, which is later used to generate the complete project files needed to build the full version.

Since GL-Get needs the Premake sources anyway in order to be built, we decided to rely on Premake to do the initial build. GL-Get injects a lot of automated functionality into Premake, but we can't rely on these features during the bootstrap process. This means we need two different ways to generate the project files: one when using Premake directly (only for bootstrapping), and another one when being built as a GL-Get package.

To simplify the maintainability we created two different Premake files which include a third one that contains the common code (as shown in Figure 25). Premake (and GL-Get by extension) run a file called *premake5.lua* by default, but a custom filename can be specified, so *bootstrap.lua* is given to Premake to generate the basic project files.

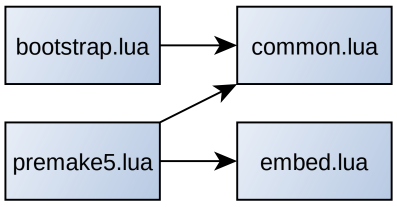


Figure 25: GL-Get's Premake files

5.4. Bootstrapping

We can use Premake to embed GL-Get's scripts, so we can avoid implementing a custom mode to work without the embedded scripts. We made our embedded script packager based on Premake's one. The main *premake5.lua* file directly delegates to *embed.lua* when running the *embed* action, so it can be run directly with Premake without specifying a Premake file.

After we generated the embedded scripts and basic project files, we can build an initial version of GL-Get. Once we have it, we can use it to build GL-Get as a standard package, by running the update, generate and build GL-Get commands on it.

The full bootstrapping process is depicted in Figure 26.

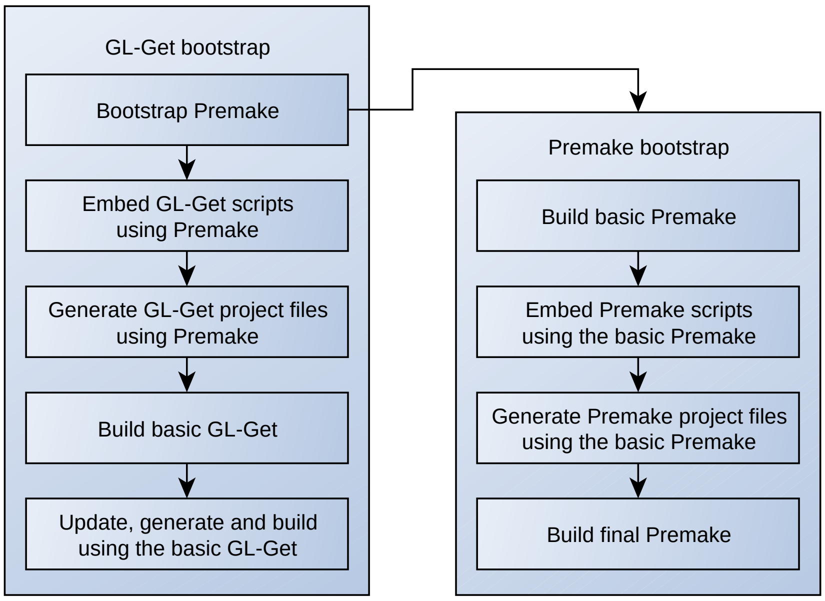

GL-Get bootstrap
Bootstrap Premake
Embed GL-Get scripts using Premake
Generate GL-Get project files using Premake
Build basic GL-Get
Update, generate and build using the basic GL-Get
Premake bootstrap
Build basic Premake
Embed Premake scripts using the basic Premake
Generate Premake project files using the basic Premake
Build final Premake

Figure 26: The GL-Get bootstrap process

5.5. External libraries

In order to help and accelerate the development process, several third party libraries were used:

- Boost [17]: A set of C++ libraries. We just use the program_options library for the command line options parsing module specified in the 4.3.1 section.
- compat53 [120]: Compatibility module providing Lua-5.3-style APIs for Lua 5.2 and 5.1.
- Penlight [117]: A set of Lua libraries that contain many common code patterns, like a class system, comprehension lists, file-system operations, printing of tables, string and table manipulation helpers, and many others.
- LuaFileSystem [119]: File System Library for Lua, used by Penlight. It's implemented in C to be able to access operating system's specific features.
- tableshape [121]: A type checking library that allows checking for the structure of tables.
- typecheck [122]: Gradual type checking for Lua functions.

- ansicolors [123]: ANSI terminal color manipulation for Lua.
- Jeffrey Friedl's JSON [124]: Simple JSON Encode/Decode in pure Lua.

5.6. Tests

Using a dynamically typed language it was vital to have a good test suite to make sure no regressions are introduced. A test suite allows writing automated tests once that can easily be run at any moment to verify everything's working as expected.

When writing a new module or a new functionality, we tried to write new tests, both to make sure it worked as expected, and also to make sure we don't break it in the future.

When possible (when the expected results were known beforehand and the process of writing the test didn't slow the development process too much), we tried to use test driven development (TDD). TDD is based on the idea that the tests should be written before the actual code, based on the function specs, considering the development finished once the tests pass.

For the actual implementation of the test suites we used the Telescope [125] Lua framework.

We implemented two main test suites: a unit tests one and an executable tests one.

- The unit test suite loads Lua modules and directly checks the behavior of individual modules or functions.
- The executable test suite uses the compiled system and checks its behavior as a black box. It uses many combinations of command-line arguments to verify proper error messages are given and the system works as expected when given the proper arguments.

After running the test suites, a report is generated. It's specially useful when run on the continuous integration system, since it gives a global vision on the evolution of the test results.

6. Planning and costs

An important factor on every development is the economic analysis. Everything has a cost, and the development of this project should also be valued.

The academic project has a charge of 37,5 credits, with an estimation of 20 hours per credit, which would give a result of 750 hours.

6.1. Planning

Since the initial requirements were vague and the development environment was unknown for us, we couldn't do proper planning beforehand. We faced the project with an iterative approach: we worked on what was needed at the moment, following the priorities dictated by the project lead and changing the goals on demand.

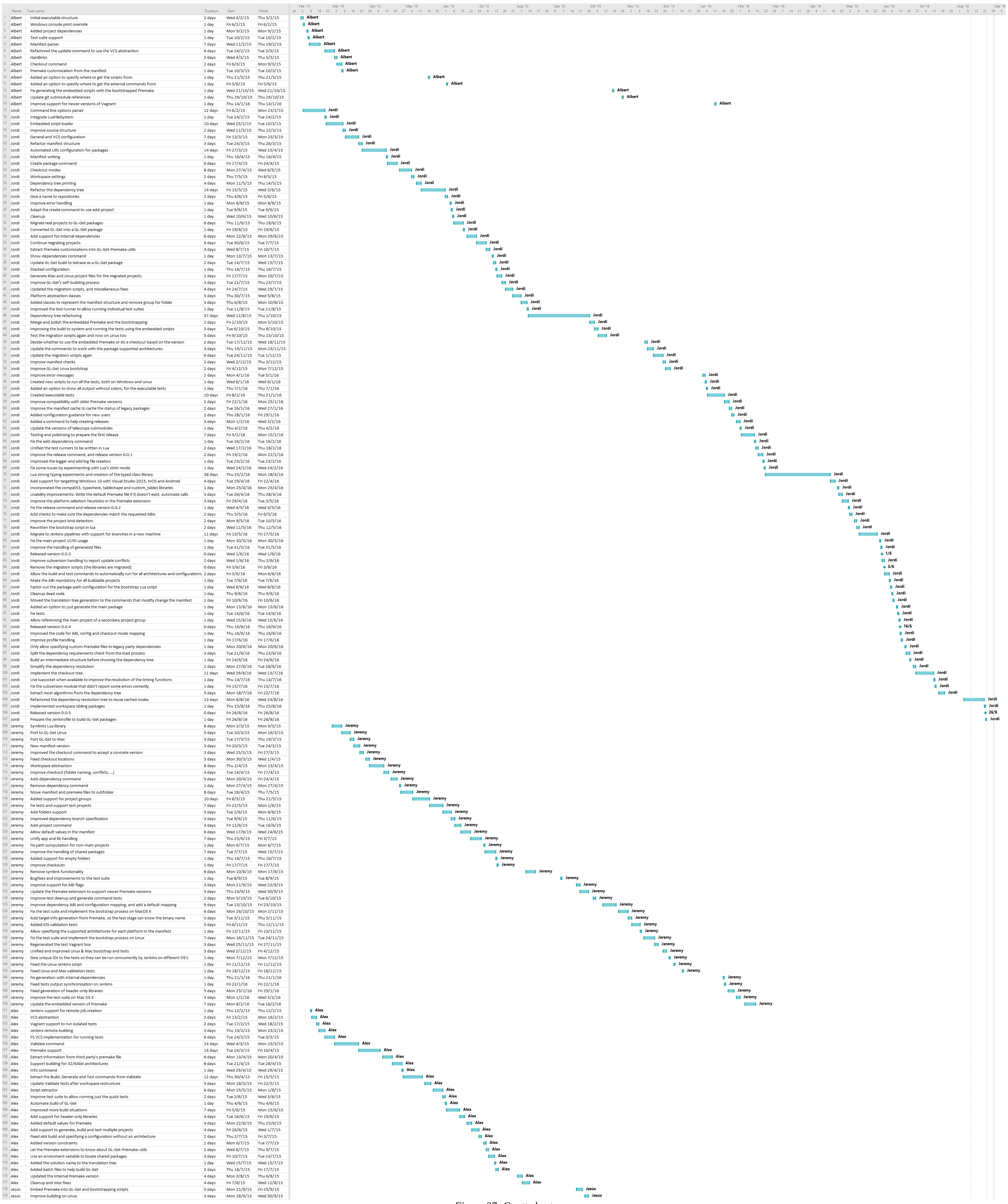
In the end, we can only do a retrospective showing how the time was spent. Figure 27 shows a Gantt chart that summarizes the history on the code repository, grouped by developer.

At the beginning of the project, the priorities were mainly filling the holes, adding new features. Later on, we started suffering the lack of design and we had to do some heavy refactors. On the last stages, the main priority was to support the first system users, by fixing bugs and implementing some of their proposed improvements.

From this information I've further grouped the tasks and developed the table 5, which breaks down the Gantt chart and easily shows who worked on each of the main tasks done during the project, and their total time spent on the project.

	Albert	Jordi	Jeremy	Alex	Jesús
Features & modules					
Standalone executable	48	0	0	24	0
Manifest	56	112	408	0	0
Workspace	0	24	64	0	0
Shared folder	16	0	152	24	0
Options parser	0	6	0	0	0
Embedded scripts	8	80	0	0	0
Script extractor	0	0	0	48	0
Premake integration	8	32	56	144	20
Remote build using Jenkins	0	0	0	32	0
VCS abstraction	0	208	0	64	0
Platform abstraction	0	88	8	0	0
ABI flags	0	32	96	0	0
Dependency tree	0	688	0	16	0
Checkout tree	0	88	0	0	0
Premake extension	0	16	40	88	0
Premake-utils	0	24	0	8	0
Logger	0	8	0	0	0
Target platforms	0	32	40	64	0
General improvements	0	168	0	32	0
Commands / Use cases					
Create package command	0	56	0	0	0
Checkout command	16	64	88	0	0
Update command	48	8	0	0	0
Generate command	0	8	64	56	0
Build command	0	8	0	104	0
Test command	0	8	24	40	0
Add-project command	0	0	24	40	0
Add-dependency command	0	8	40	0	0
Remove-dependency command	0	0	8	0	0
Validate command	0	0	8	176	0
Show-dependencies command	0	8	0	0	0
Info command	0	0	0	8	0
Infrastructure & Management					
Bootstrapping	0	56	24	0	20
Test suite	16	152	32	32	0
Dependencies management	8	24	0	0	0
Jenkins management	0	88	24	0	0
Release creation	0	104	0	0	0
Linux port	0	0	96	0	24
MacOS X port	0	0	96	0	0
Migration of the existing projects	0	240	0	0	0
Strong typing for Lua	0	312	0	0	0
TOTAL	**224**	**2840**	**1392**	**960**	**64**

Table 5: Summary of the Gantt chart (hours spent on tasks per developer)



	Name	Task name	Duration	Start	Finish
1	Albert	Initial executable structure	2 days	Wed 4/2/15	Thu 5/2/15
2	Albert	Windows console print override	1 day	Fri 6/2/15	Fri 6/2/15
3	Albert	Added project dependencies	1 day	Mon 9/2/15	Mon 9/2/15
4	Albert	Test suite support	1 day	Tue 10/2/15	Tue 10/2/15
5	Albert	Manifest parser	7 days	Wed 11/2/15	Thu 19/2/15
6	Albert	Refactored the update command to use the VCS abstraction	6 days	Tue 24/2/15	Tue 3/3/15
7	Albert	Hardlinks	2 days	Wed 4/3/15	Thu 5/3/15
8	Albert	Checkout command	2 days	Fri 6/3/15	Mon 9/3/15
9	Albert	Premake customization from the manifest	1 day	Tue 10/3/15	Tue 10/3/15
10	Albert	Added an option to specify where to get the scripts from	1 day	Thu 21/5/15	Thu 21/5/15
11	Albert	Added an option to specify where to get the external commands from	1 day	Fri 5/6/15	Fri 5/6/15
12	Albert	Fix generating the embedded scripts with the bootstrapped Premake	1 day	Wed 21/10/15	Wed 21/10/15
13	Albert	Update git submodule references	1 day	Thu 29/10/15	Thu 29/10/15
14	Albert	Improve support for newer versions of Vagrant	1 day	Thu 14/1/16	Thu 14/1/16
15	Jordi	Command line options parser	12 days	Fri 6/2/15	Mon 23/2/15
16	Jordi	Integrate LuaFileSystem	1 day	Tue 24/2/15	Tue 24/2/15
17	Jordi	Embedded script loader	10 days	Wed 25/2/15	Tue 10/3/15
18	Jordi	Improve source structure	2 days	Wed 11/3/15	Thu 12/3/15
19	Jordi	General and VCS configuration	7 days	Fri 13/3/15	Mon 23/3/15
20	Jordi	Refactor manifest structure	3 days	Tue 24/3/15	Thu 26/3/15
21	Jordi	Automated URL configuration for packages	14 days	Fri 27/3/15	Wed 15/4/15
22	Jordi	Manifest writing	1 day	Thu 16/4/15	Thu 16/4/15
23	Jordi	Create package command	6 days	Fri 17/4/15	Fri 24/4/15
24	Jordi	Checkout modes	8 days	Mon 27/4/15	Wed 6/5/15
25	Jordi	Workspace settings	2 days	Thu 7/5/15	Fri 8/5/15
26	Jordi	Dependency tree printing	4 days	Mon 11/5/15	Thu 14/5/15
27	Jordi	Refactor the dependency tree	14 days	Fri 15/5/15	Wed 3/6/15
28	Jordi	Give a name to repositories	2 days	Thu 4/6/15	Fri 5/6/15
29	Jordi	Improve error handling	1 day	Mon 8/6/15	Mon 8/6/15
30	Jordi	Adapt the create command to use add-project	1 day	Tue 9/6/15	Tue 9/6/15
31	Jordi	Cleanup	1 day	Wed 10/6/15	Wed 10/6/15
32	Jordi	Migrate real projects to GL-Get packages	6 days	Thu 11/6/15	Thu 18/6/15
33	Jordi	Converted GL-Get into a GL-Get package	1 day	Fri 19/6/15	Fri 19/6/15
34	Jordi	Add support for internal dependencies	6 days	Mon 22/6/15	Mon 29/6/15
35	Jordi	Continue migrating projects	6 days	Tue 30/6/15	Tue 7/7/15
36	Jordi	Extract Premake customizations into GL-Get-Premake-utils	3 days	Wed 8/7/15	Fri 10/7/15
37	Jordi	Show-dependencies command	1 day	Mon 13/7/15	Mon 13/7/15
38	Jordi	Update GL-Get build to behave as a GL-Get package	2 days	Tue 14/7/15	Wed 15/7/15
39	Jordi	Stacked configuration	1 day	Thu 16/7/15	Thu 16/7/15
40	Jordi	Generate Mac and Linux project files for the migrated projects	2 days	Fri 17/7/15	Mon 20/7/15
41	Jordi	Improve GL-Get's self-building process	3 days	Tue 21/7/15	Thu 23/7/15
42	Jordi	Updated the migration scripts, and miscellaneous fixes	4 days	Fri 24/7/15	Wed 29/7/15
43	Jordi	Platform abstraction classes	5 days	Thu 30/7/15	Wed 5/8/15
44	Jordi	Added classes to represent the manifest structure and remove group for folder	3 days	Thu 6/8/15	Mon 10/8/15
45	Jordi	Improved the test runner to allow running individual test suites	1 day	Tue 11/8/15	Tue 11/8/15
46	Jordi	Dependency tree refactoring	37 days	Wed 12/8/15	Thu 1/10/15
47	Jordi	Merge and polish the embedded Premake and the bootstrapping	2 days	Fri 2/10/15	Mon 5/10/15
48	Jordi	Improving the build to system and running the tests using the embedded scripts	3 days	Tue 6/10/15	Thu 8/10/15
49	Jordi	Test the migration scripts again and now on Linux too	5 days	Fri 9/10/15	Thu 15/10/15
50	Jordi	Decide whether to use the embedded Premake or do a checkout based on the version	2 days	Tue 17/11/15	Wed 18/11/15
51	Jordi	Update the commands to work with the package supported architectures	3 days	Thu 19/11/15	Mon 23/11/15
52	Jordi	Update the migration scripts again	6 days	Tue 24/11/15	Tue 1/12/15
53	Jordi	Improve manifest checks	2 days	Wed 2/12/15	Thu 3/12/15
54	Jordi	Improve GL-Get Linux bootstrap	2 days	Fri 4/12/15	Mon 7/12/15
55	Jordi	Improve error messages	2 days	Mon 4/1/16	Tue 5/1/16
56	Jordi	Created new scripts to run all the tests, both on Windows and Linux	1 day	Wed 6/1/16	Wed 6/1/16
57	Jordi	Added an option to show all output without colors, for the executable tests	1 day	Thu 7/1/16	Thu 7/1/16
58	Jordi	Created executable tests	10 days	Fri 8/1/16	Thu 21/1/16
59	Jordi	Improve compatibility with older Premake versions	2 days	Fri 22/1/16	Mon 25/1/16
60	Jordi	Improve the manifest cache to cache the status of legacy packages	2 days	Tue 26/1/16	Wed 27/1/16
61	Jordi	Added configuration guidance for new users	2 days	Thu 28/1/16	Fri 29/1/16
62	Jordi	Added a command to help creating releases	3 days	Mon 1/2/16	Wed 3/2/16
63	Jordi	Update the versions of telescope submodules	1 day	Thu 4/2/16	Thu 4/2/16
64	Jordi	Testing and polishsing to prepare the first release	7 days	Fri 5/2/16	Mon 15/2/16
65	Jordi	Fix the add-dependency command	1 day	Tue 16/2/16	Tue 16/2/16
66	Jordi	Unified the test runners to be written in Lua	2 days	Wed 17/2/16	Thu 18/2/16
67	Jordi	Improve the release command, and release version 0.0.1	2 days	Fri 19/2/16	Mon 22/2/16
68	Jordi	Improved the logger and add log file creation	1 day	Tue 23/2/16	Tue 23/2/16
69	Jordi	Fix some issues by experimenting with Lua's strict mode	1 day	Wed 24/2/16	Wed 24/2/16
70	Jordi	Lua strong typing experiments and creation of the typed class library	38 days	Thu 25/2/16	Mon 18/4/16
71	Jordi	Add support for targetting Windows 10 with Visual Studio 2015, tvOS and Android	4 days	Tue 19/4/16	Fri 22/4/16
72	Jordi	Incorporated the compat53, typecheck, tableshape and custom_tables libraries	1 day	Mon 25/4/16	Mon 25/4/16
73	Jordi	Usability improvements: Write the default Premake file if it doesn't exist, automatic calls	3 days	Tue 26/4/16	Thu 28/4/16
74	Jordi	Improve the platform selection heuristics in the Premake extension	3 days	Fri 29/4/16	Tue 3/5/16
75	Jordi	Fix the release command and release version 0.0.2	1 day	Wed 4/5/16	Wed 4/5/16
76	Jordi	Add checks to make sure the dependencies match the requested ABIs	2 days	Thu 5/5/16	Fri 6/5/16
77	Jordi	Improve the project kind detection	2 days	Mon 9/5/16	Tue 10/5/16
78	Jordi	Rewritten the bootstrap script in lua	2 days	Wed 11/5/16	Thu 12/5/16
79	Jordi	Migrate to Jenkins pipelines with support for branches in a new machine	11 days	Fri 13/5/16	Fri 27/5/16
80	Jordi	Fix the main project UUID usage	1 day	Mon 30/5/16	Mon 30/5/16
81	Jordi	Improve the handling of generated files	1 day	Tue 31/5/16	Tue 31/5/16
82	Jordi	Released version 0.0.3	0 days	Wed 1/6/16	Wed 1/6/16
83	Jordi	Improve subversion handling to report update conflicts	2 days	Wed 1/6/16	Thu 2/6/16
84	Jordi	Remove the migration scripts (the libraries are migrated)	0 days	Fri 3/6/16	Fri 3/6/16
85	Jordi	Allow the build and test commands to automatically run for all architectures and configurations	2 days	Fri 3/6/16	Mon 6/6/16
86	Jordi	Make the ABI mandatory for all buildable projects	1 day	Tue 7/6/16	Tue 7/6/16
87	Jordi	Factor out the package-path configuration for the bootstrap Lua script	1 day	Wed 8/6/16	Wed 8/6/16
88	Jordi	Cleanup dead code	1 day	Thu 9/6/16	Thu 9/6/16
89	Jordi	Moved the translation tree generation to the commands that modify change the manifest	1 day	Fri 10/6/16	Fri 10/6/16
90	Jordi	Added an option to just generate the main package	1 day	Mon 13/6/16	Mon 13/6/16
91	Jordi	Fix tests	1 day	Tue 14/6/16	Tue 14/6/16
92	Jordi	Allow referencing the main project of a secondary project group	1 day	Wed 15/6/16	Wed 15/6/16
93	Jordi	Released version 0.0.4	0 days	Thu 16/6/16	Thu 16/6/16
94	Jordi	Improved the code for ABI, config and checkout mode mapping	1 day	Thu 16/6/16	Thu 16/6/16
95	Jordi	Improve profile handling	1 day	Fri 17/6/16	Fri 17/6/16
96	Jordi	Only allow specifying custom Premake files to legacy party dependencies	1 day	Mon 20/6/16	Mon 20/6/16
97	Jordi	Split the dependency requirements check from the load process	3 days	Tue 21/6/16	Thu 23/6/16
98	Jordi	Build an intermediate structure before showing the dependency tree	1 day	Fri 24/6/16	Fri 24/6/16
99	Jordi	Simplify the dependency resolution	2 days	Mon 27/6/16	Tue 28/6/16
100	Jordi	Implement the checkout tree	11 days	Wed 29/6/16	Wed 13/7/16
101	Jordi	Use luasocket when available to improve the resolution of the timing functions	1 day	Thu 14/7/16	Thu 14/7/16
102	Jordi	Fix the subversion module that didn't report some errors correctly	1 day	Fri 15/7/16	Fri 15/7/16
103	Jordi	Extract most algorithms from the dependency tree	5 days	Mon 18/7/16	Fri 22/7/16
104	Jordi	Refactored the dependency resolution tree to reuse cached nodes	13 days	Mon 8/8/16	Wed 24/8/16
105	Jordi	Implemented workspace sibling packages	1 day	Thu 25/8/16	Thu 25/8/16
106	Jordi	Released version 0.0.5	0 days	Fri 26/8/16	Fri 26/8/16
107	Jordi	Prepare the Jenkinsfile to build GL-Get packages	1 day	Fri 26/8/16	Fri 26/8/16
108	Jeremy	Symlinks Lua library	6 days	Mon 2/3/15	Mon 9/3/15
109	Jeremy	Port to GL-Get Linux	5 days	Tue 10/3/15	Mon 16/3/15
110	Jeremy	Port GL-Get to Mac	3 days	Tue 17/3/15	Thu 19/3/15
111	Jeremy	New manifest version	3 days	Fri 20/3/15	Tue 24/3/15
112	Jeremy	Improved the checkout command to accept a concrete version	3 days	Wed 25/3/15	Fri 27/3/15
113	Jeremy	Fixed checkout locations	3 days	Mon 30/3/15	Wed 1/4/15
114	Jeremy	Workspace abstraction	8 days	Thu 2/4/15	Mon 13/4/15
115	Jeremy	Improve checkout (folder naming, conflicts, ...)	4 days	Tue 14/4/15	Fri 17/4/15
116	Jeremy	Add-dependency command	5 days	Mon 20/4/15	Fri 24/4/15
117	Jeremy	Remove-dependency command	1 day	Mon 27/4/15	Mon 27/4/15
118	Jeremy	Move manifest and premake files to subfolder	8 days	Tue 28/4/15	Thu 7/5/15
119	Jeremy	Added support for project groups	10 days	Fri 8/5/15	Thu 21/5/15
120	Jeremy	Fix tests and support test projects	7 days	Fri 22/5/15	Mon 1/6/15
121	Jeremy	Add folders support	5 days	Tue 2/6/15	Mon 8/6/15
122	Jeremy	Improved dependency branch specification	3 days	Tue 9/6/15	Thu 11/6/15
123	Jeremy	Add-project command	3 days	Fri 12/6/15	Tue 16/6/15
124	Jeremy	Allow default values in the manifest	6 days	Wed 17/6/15	Wed 24/6/15
125	Jeremy	Unify app and lib handling	7 days	Thu 25/6/15	Fri 3/7/15
126	Jeremy	Fix path computation for non-main projects	1 day	Mon 6/7/15	Mon 6/7/15
127	Jeremy	Improve the handling of shared packages	7 days	Tue 7/7/15	Wed 15/7/15
128	Jeremy	Added support for empty folders	1 day	Thu 16/7/15	Thu 16/7/15
129	Jeremy	Improve checkouts	1 day	Fri 17/7/15	Fri 17/7/15
130	Jeremy	Remove symlink functionality	6 days	Mon 10/8/15	Mon 17/8/15
131	Jeremy	Bugfixes and improvements to the test suite	1 day	Tue 8/9/15	Tue 8/9/15
132	Jeremy	Improve support for ABI flags	3 days	Mon 21/9/15	Wed 23/9/15
133	Jeremy	Update the Premake extension to support newer Premake versions	5 days	Thu 24/9/15	Wed 30/9/15
134	Jeremy	Improve test cleanup and generate command tests	2 days	Mon 5/10/15	Tue 6/10/15
135	Jeremy	Improve dependency ABI and configuration mapping, and add a default mapping	9 days	Tue 13/10/15	Fri 23/10/15
136	Jeremy	Fix the test suite and implement the bootstrap process on MacOS X	6 days	Mon 26/10/15	Mon 2/11/15
137	Jeremy	Add target info generation from Premake, so the test stage can know the binary name	3 days	Tue 3/11/15	Thu 5/11/15
138	Jeremy	Added iOS validation tests	5 days	Fri 6/11/15	Thu 12/11/15
139	Jeremy	Allow specifying the supported architectures for each platform in the manifest	1 day	Fri 13/11/15	Fri 13/11/15
140	Jeremy	Fix the test suite and implement the bootstrap process on Linux	7 days	Mon 16/11/15	Tue 24/11/15
141	Jeremy	Regenerated the test Vagrant box	3 days	Wed 25/11/15	Fri 27/11/15
142	Jeremy	Unified and improved Linux & Mac bootstrap and tests	3 days	Wed 2/12/15	Fri 4/12/15
143	Jeremy	Give unique IDs to the tests so they can be run concurrently by Jenkins on different OS's	1 day	Mon 7/12/15	Mon 7/12/15
144	Jeremy	Fixed the Linux Jenkins script	1 day	Fri 11/12/15	Fri 11/12/15
145	Jeremy	Fixed Linux and Mac validation tests	1 day	Fri 18/12/15	Fri 18/12/15
146	Jeremy	Fix generation with internal dependencies	1 day	Thu 21/1/16	Thu 21/1/16
147	Jeremy	Fixed tests output synchronization on Jenkins	1 day	Fri 22/1/16	Fri 22/1/16
148	Jeremy	Fixed generation of header-only libraries	5 days	Mon 25/1/16	Fri 29/1/16
149	Jeremy	Improve the test suite on Mac OS X	3 days	Mon 1/2/16	Wed 3/2/16
150	Jeremy	Update the embedded version of Premake	7 days	Mon 8/2/16	Tue 16/2/16
151	Alex	Jenkins support for remote job creation	1 day	Thu 12/2/15	Thu 12/2/15
152	Alex	VCS abstraction	2 days	Fri 13/2/15	Mon 16/2/15
153	Alex	Vagrant support to run isolated tests	2 days	Tue 17/2/15	Wed 18/2/15
154	Alex	Jenkins remote building	3 days	Thu 19/2/15	Mon 23/2/15
155	Alex	FS VCS implementation for running tests	6 days	Tue 24/2/15	Tue 3/3/15
156	Alex	Validate command	14 days	Wed 4/3/15	Mon 23/3/15
157	Alex	Premake support	14 days	Tue 24/3/15	Fri 10/4/15
158	Alex	Extract information from third party's premake file	6 days	Mon 13/4/15	Mon 20/4/15
159	Alex	Support building for 32/64bit architectures	6 days	Tue 21/4/15	Tue 28/4/15
160	Alex	Info command	1 day	Wed 29/4/15	Wed 29/4/15
161	Alex	Extract the Build, Generate and Test commands from Validate	12 days	Thu 30/4/15	Fri 15/5/15
162	Alex	Update Validate tests after workspace restructure	5 days	Mon 18/5/15	Fri 22/5/15
163	Alex	Script extractor	6 days	Mon 25/5/15	Mon 1/6/15
164	Alex	Improve test suite to allow running just the quick tests	2 days	Tue 2/6/15	Wed 3/6/15
165	Alex	Automate build of GL-Get	1 day	Thu 4/6/15	Thu 4/6/15
166	Alex	Improved more build situations	7 days	Fri 5/6/15	Mon 15/6/15
167	Alex	Add support for header-only libraries	4 days	Tue 16/6/15	Fri 19/6/15
168	Alex	Added default values for Premake	4 days	Mon 22/6/15	Thu 25/6/15
169	Alex	Add support to generate, build and test multiple projects	4 days	Fri 26/6/15	Wed 1/7/15
170	Alex	Fixed x64 build and specifying a configuration without an architecture	2 days	Thu 2/7/15	Fri 3/7/15
171	Alex	Added version constraints	2 days	Mon 6/7/15	Tue 7/7/15
172	Alex	Let the Premake extensions to know about GL-Get-Premake-utils	2 days	Wed 8/7/15	Thu 9/7/15
173	Alex	Use an enviroment variable to locate shared packages	3 days	Fri 10/7/15	Tue 14/7/15
174	Alex	Added the solution name to the translation tree	1 day	Wed 15/7/15	Wed 15/7/15
175	Alex	Added batch files to help build GL-Get	2 days	Thu 16/7/15	Fri 17/7/15
176	Alex	Updated the internal Premake version	4 days	Mon 3/8/15	Thu 6/8/15
177	Alex	Cleanup and misc fixes	4 days	Fri 7/8/15	Wed 12/8/15
178	Jesús	Embed Premake into GL-Get and bootstrapping scripts	5 days	Mon 21/9/15	Fri 25/9/15
179	Jesús	Improve building on Linux	3 days	Mon 28/9/15	Wed 30/9/15

Figure 27: Gantt chart

6.2. Costs

There are many costs that could be attributed to the development of this project. The most important cost in this project are the human resources, but there are other derived costs. These are detailed in the following sections.

6.2.1. Human resources

Every hour spent on the development should be valued as an effort and needs a compensation. We consider each role to have a different wage, as detailed on table 6. These are estimated as the rough cost for the company.

Role	Hourly wage
Project lead	35€
Developer	20€

Table 6: Hourly wages per role

Based on the table above and the hours detailed on the planning section, we can easily compute the total cost attributable to human resources, as shown on table 7.

Developer	Hourly wage	Hours	Subtotal
Albert	35€	224h	7840€
Jordi	20€	2840h	56 800€
Jeremy	20€	1392h	27 840€
Alex	20€	960h	19 200€
Jesús	20€	64h	1280€
Total			**112 960€**

Table 7: Human resource costs

The total cost of human resources sums 112 960€.

6.2.2. Other costs

Other than the human resources, the development of any software project also has other costs that may not be so evident at first. First, there's the hardware costs. While no hardware was bought exclusively for the development of this project, some existing hardware was used. For this we'll count the amortization for the duration of the project:

- A standard desktop computer was used, with an estimated price of 1000€. Its estimated useful lifetime is around 5 years, which gives an amortization cost of 200€ per year.

The used software licenses should also be taken into account, using the same amortization criteria as for the hardware:

- Microsoft Visual Studio Professional 2013 was used. The 2015 version has a price of 646€ on the Microsoft Store, so we'll conclude the price of the previous version was similar. Since a new version is launched every 2-3 years, we'll consider its useful lifetime around 3 years, giving a yearly amortization cost of 215€.

The total amortization costs are computed in table 8, based on a yearly amortization cost per developer of 200€ of hardware and 215€ of software. The years spent per developer on the project are approximations based on the data from the previous section. After this we get a total of 610€ of hardware and 655,75€ of software costs.

Developer	Years	Hardware amortization	Software amortization
Albert	0,2	40€	43€
Jordi	1,5	300€	322,5€
Jeremy	0,8	160€	172€
Alex	0,5	100€	107,5€
Jesús	0,05	10€	10,75€
Subtotals		**610€**	**655,75€**

Table 8: Other costs

We could also include other costs, like the Internet access or the power bills, but since these services are shared among a lot of coworkers on the same office, we'll consider these costs to be irrelevant compared to the previous ones.

6.2.3. Total cost computation

Adding the costs elaborated on the previous sections we can get the total cost.

Description	Subtotal
Human resources	112 960€
Hardware amortization	610€
Software licenses	655,75€
Total	**114 225,75€**

Table 9: Computation of the total development cost

As shown on table 9, the total cost of development of this project has been 114 225,75€.

7. Conclusions

To conclude the memory of this project, the current chapter gives an overview of the current status of the project, what went well, what went bad, and how it could be improved in the future.

7.1. Achieved goals

The system is functional, and it has already been in use in production by some developers in the Barcelona studio for a few months.

There's around 10 to 15 core libraries already migrated to follow GL-Get's conventions, and some of them are actively developed with the help of the tool. This also includes some third party libraries like OpenSSL, cURL, LZ4, Zlib and RapidJSON, for which we maintain internal copies, and we also converted them to GL-Get packages. Now we can simply "depend on cURL version 7.49.1", without having to care about whether we also have to link with OpenSSL or not.

The current implementation covers most situations of most use cases. Some corner cases may not be covered completely, and in some cases an error could be triggered, but there are workarounds for most situations. I continue working on the project to polish it and fix the bugs as they arise.

Some use cases, like the "publish", "branch" or "merge" ones haven't been implemented yet, but there's no impediment to do the process manually (modifying the manifest accordingly to reflect the new branch or version and doing the relevant VCS action).

7.2. Problems found

During the development of the system, we faced several pitfalls, and some things could have been done better, looking in retrospective.

7.2.1. Lack of design

When we started working on the project, we just had the prototype and a few general directions. The structure of the manifest wasn't well defined until many months later, when we already had worked on several features (for example, at the beginning there were no project groups, the manifest just contained a single project, etc). The design changed often during the development, and it was hard to have a clear idea of the final requirements, which slowed the process.

Iterating on top of a prototype didn't help either, since its implementation was lacking in many aspects (it was done as a quick proof of concept). It was based on several preconceived ideas that may not have made the final system.

While some parts of the original implementation were okay, many others didn't meet the standards of the rest of the application, and we had to rewrite several parts of it. On the other hand, some parts of the original code are still present, because we couldn't justify the time to rewrite them.

7.2.2. Lua

When we started working on the project, the whole team was fairly new to the Lua language, and the lack of knowledge on the language caused us some headaches. As an example, we hadn't agreed on some things as basic as error handling, which in the Lua ecosystem already have some conventions.

Lua is a dynamic language and, as such, it has some good features, but it also has its drawbacks. Being interpreted, was very useful for quick iteration. You could test your changes just after saving them, without having to recompile anything. On the other hand, from my point of view, one of its biggest flaws it the lack of static typing. It's very easy for the developer to introduce bugs that could go unnoticed for a long time.

As a simple example, have a look at the following Lua snippet:

```lua
function bad()
    return non_existing_table.non_existing_field
end

local MyVar = true
if myvar then
    bad()
end

print("Everything's fine")
```

Running this code would show "Everything's fine".

It is clear to any minimally experienced Lua developer to see that line 2 is incorrect because it tries to index a non-existing table. But it's valid code. The issue would only be raised when the code is executed. And why isn't it executed if there's a call to the *bad()* function? Because Lua evaluates *myvar*, which doesn't exist (it has nothing to do with *MyVar*), to *nil*, making the condition *false*. It's a common example of how Lua makes it hard to find errors. It isn't an error to misspell a variable name. And it isn't an error to have erroneous code if it isn't executed.

The dynamic nature of the language lets it rely on the side effects that could result of running other modules. So, if before running the previous code snippet we ran a module which set *myvar* to *true*, the code shown before would actually crash with the error "Attempt to index a nil value (non_existing_field not present in nil)". If additionally another module sets the global *non_existing_table* to an empty table, the code above would now be completely correct.

To summarize, I think the language is too vulnerable to global interactions. While on small scripts like the previous example it's very easy to debug, its effects in larger applications, like GL-Get, easily get in the way. It's interesting to have access to the features that the dynamic languages offer, but having it as the default is too error prone.

I think this can be gradually solved with future work, but the effort isn't small.

7.3. Future work

While the application is functional, there are always ways to improve it, either by adding new features or by improving its internals. The following is a list of a few ideas for future work.

7.3.1. Web service

With the current implementation, we can only do the package resolution in one direction: from a package to its dependencies. It would be interesting to do the reverse resolution: which packages depend on this one?

For that we would need a centralized package register (basically a database) which allows us to keep package versions information together with their dependencies. With that infrastructure in place, the publishing action would register the new releases to the package register and, for example, we could trigger automated emails to the owners of packages that depend on it ("Hey, package X just got a new release. As the maintainer of package Y you may be interested in updating your dependency").

It would also be interesting to store compatibility information of particular versions. It would allow knowing which particular versions of a dependency work well or break the compatibility. The duty of filling this compatibility database could be given to a build server that would try to build each packages with all possible version combinations of its dependencies, and send the result to the server.

7.3.2. Add static typing

As already justified in section 7.2.2, I think the lack of static typing in the Lua programming language is a big issue for the development of non-trivial systems like GL-Get. To solve this, I think it would be worth trying to add static type checking to the system. An option would be to rewrite parts of it in another language, like C/C++, or another one that also runs in the Lua VM, but it's a costly process. If Lua continues to be the main language for the system, several approaches could be used:

- Adding run-time checks
    - The usual way to check the argument types of a function is to add asserts at the beginning of the function. If we also want to check the return values, we also have to add these assertions before each return statement:

```lua
local function my_function(a, b)
  assert(type(a) == "number")
  assert(type(b) == "number")
  local result = a + b
  assert(type(result) == "number")
  return result
end
```

Fortunately, there are some libraries which simplify this task. For instance, the typecheck library [122] provides a convenient way to annotate functions to check its arguments and return types, by creating a wrapper function that does the actual checks as shown in the example above:

```
local argscheck = require "typecheck".argscheck

local my_function = argscheck "my_function (int, int) => int" ..
function (a, b)
  return a + b
end
```

- In Lua most values are stored in tables. Even the global variables are accessed through the _G table. Lua allows plugging additional functionality into tables with the use of metatables (as explained in the 5.1.3 section). With this in mind and using several tricks, during the project I created a library called custom_tables, which allows creating tables with several custom features:
  - Proxy tables, which get and set the values from another table. It has the advantage that we can get events even when accessing existing fields.
  - Getter and setter chaining.
  - Read-only tables.
  - Strict tables, which throw an error when trying to read a field that hasn't been set yet.
  - Auto-tables, which create non-existing fields when being accessed, allowing easy setting of deep-nested fields.
  - Declared tables, which take a field declaration list and only allow setting or getting fields from this list.
  - Typed tables, which are in addition strict and declared tables, and take a list of field type declaration, and verify that all field sets are of the proper type.
  - Typed classes, based on Penlight's class and the typed tables above.

  This functionality has already been adopted lately in the project, and several classes have been created using it, already showing that it helps debugging real world code. The checks on the typed class are currently limited to the fields. It would be very nice to extend its features to also check the method types, and to automatically propagate these checks to the inherited classes.

  - Local variables are an open issue, because Lua doesn't feature triggers to intercept the getting or setting of local variables, but since their scope is more limited and they can't easily be accessed from the outside of their scope, it's not such a big deal.

- Adding compile-time checks
  - MoonScript [126]: It's a scripting language that compiles to Lua and offers some extra features, like native classes, but it's still dynamically typed, like Lua, so we probably need something more.
  - TypedLua [127]: It's an in-development language which is a super-set of Lua (all Lua code is valid Typed Lua), and it compiles to plain Lua code. It basically allows adding type annotations to the code, in variable declarations, function arguments or function return values. It also has limited support for custom table types. When the compiler is run, it checks for type matching and it does a great job even at type inference. During the project I evaluated the possibility of adopting it, and it looks promising, but the implementation is still a bit incomplete.

    Anyway, being based on Lua, and being completely interoperable with it, at compile time we can't make sure that the functions will be called properly from external Lua code, so a mixed solution with run-time checks would be ideal. I started implementing a process that injected run-time checks into the Typed Lua compiler. The generated code used the previously cited libraries for run-time checks (typecheck for function types and custom_tables for tables), but I didn't have time to finish the implementation and it didn't reach production code.
  - Metalua [128]: It's a framework for building custom functionality into the Lua language, implementing it using Lua itself. Their tutorials show a few examples of how to add type checking and native class functionality to the language. It could be worth checking, but I don't know how compatible it would be with the existing code.

7.3.3. Add version compatibility checks

A question that's common for developers is "what's a proper version number for the next release?". Version numbers are often assigned on how the developers feel about the importance of a specific release, but it shouldn't be like this. For example, there's some common rules for library versioning (which were the base for the Semantic Versioning [115] standard):

- Changes that break the existing API should raise the major version, even if no new feature is added.
- New features that consist of API additions which don't break the existing API should raise the minor version.
- Fixes that don't modify the API at all should only raise the patch version.

While the theory is easy to understand, it's hard for the developer to make sure he adhered to the rules. There's some existing software to detect API changes, like API Diff [129], but it usually just shows some information and the developer must interpret this and take the final decision. GL-Get knows a lot of information about the packages, and this could be used to give better advice.

Most packages should contain test projects, and these should make sure the API behaves as expected in that version. In this particular case, a relatively easy solution would be to create a new command that combined the tests with the main project of two different versions of the same package:

- If the tests of the old version can be built with the main project of the new version, and can be run successfully, it would mean the newer version didn't break the API: there's no need to raise the major version.
- If the tests of the new version can be built with the main project of the old version, it would mean no new APIs were introduced: there's no need to raise the minor version.
- In addition, if the new tests run successfully with the main project of the old version, it would mean that no behavior change was detected, so it could suggest that the implementation changes weren't tested properly (regression tests should be written to make sure old bugs aren't reintroduced).

These automated checks could be used to suggest the next version number to the developer, or to warn him if he wants to publish a new release with a version number that doesn't correspond to the changes.

Combined with the web service described in section 7.3.1, we could also offer a way to easily check and visualize the compatibility of two arbitrary versions, which isn't only based on the version number.

7.3.4. Testing framework

While developing the aforementioned custom_tables library, I tried to adopt the Lua community standards, and with that I discovered another testing framework called busted [130], which seems to be the standard nowadays.

We're using Telescope [125] for GL-Get's unit and executable tests. While it's functional, it has several limitations (it's very simple), and it seems to be unmaintained. For instance, Telescope was missing an XML report file output (which I implemented), but busted already supported it, in addition to many more advanced features.

It could be worth trying to replace the testing framework to work with busted.

7.3.5. Modularization

When new functionality was needed, it was implemented directly into the application. We kept a “utils” folder for helpers and features that were not directly related to GL-Get. It contains lots of stuff, like the logging system or the VCS and build system abstractions, with an interface ad-hoc to our needs at the moment.

Many of these things could be very useful if they were properly extracted as libraries and their API was revisited to offer an interface generic enough to cover what could be needed by other projects. That would probably also improve the code quality, since the current APIs were done without much thought.

Additionally, it would be interesting to adopt the LuaRocks [72] system to manage the project dependencies. We're using several third party Lua libraries. Right now we're basically using git submodules, and keeping up to date with them is getting harder. Our libraries could also be packaged as LuaRocks and the same system could be used to fetch and update our own libraries.

7.3.6. Use a single programming language

When we started working on the project, it was our first real contact with the Lua programming language. At that time it seemed sensible to have a mix of Lua and C/C++ modules that interacted well with each other, based on our needs. We ended up having some modules, like the command line options parsing, implemented unnecessarily in C/C++. In the end, it made the implementation harder (one had to switch constantly between languages to implement new features), and these features were almost impossible to test.

It would be interesting to migrate as many modules as possible to a common language (whatever is the chosen direction, either Lua or C/C++).

7.3.7. Improve extensibility

Right now the system is very tightly coupled with Premake and it's based on strong conventions. Here are a few examples:

- The package versions must follow the Semantic Versioning [115], which is a good standard, but some existing third party projects use incompatible numberings (like OpenSSL, which puts letters attached to the numbers: 1.0.1t). For these we now try to convert their versions into a number compatible with Semantic Versioning, but we don't always achieve the desired effect.
- It would be nice to allow alternative project generation and build systems and custom test runners, like having the ability to call arbitrary commands or shell scripts. As an example, GL-Get isn't aware of its own tests (and it can't be used to run them) because they're not compiled programs, they're a set of shell and Lua scripts.
- The package structure in the file-system is very strict. The good point is that people working on the same organization will quickly get the structure of packages maintained by other developers. On the other hand, it's too restrictive to use third party or legacy packages: these have to be migrated to the GL-Get structure, which means moving or renaming files, and it forces us to have our own versions of those projects, and it makes it much harder to keep up to date with the upstream projects.
- The package manifest must reside inside the package VCS. It would be very useful to be able to specify the sources location in the manifest and the package structure, so we could create GL-Get packages for third party projects much easier. This would allow us

for example to have a repository of external projects by just storing their manifests and some helper files which would "overlay" on top of the upstream projects.

7.4. Personal conclusions

Personally, when I entered the company, I felt like the lack of package management was very frustrating, and the used solutions were very rudimentary (like using batch files to lock the revision of SVN externals...), so having the opportunity to work on this project came in handy and it was very appreciated. I think we've contributed a bit to the tranquility and sanity of software developers.

Having worked on this project for over a year and a half has been a really great experience. I've learned a lot (mainly Lua, which I've loved and I've hated, with the second one winning the battle), and I've enjoyed working with a team of amazing professionals.

Being critic, the current implementation isn't as complete as I would like, and it feels a bit rough sometimes. Anyway, overall the result is good: it can be used, and it has already made some developers happy. We can always come back to improve it.

If I could restart it from scratch, I would probably change the implementation language for something with static typing. It gets on my nerves to know there may be many typing errors currently hidden in the code, which won't be uncovered until a concrete codepath is executed.

All in all, I feel we're in the good direction and I hope we can continue polishing it until we reach a stage of maturity where it would make sense to release it outside of Gameloft. I think it could be very helpful for a lot of developers out there, and the company seems open to the possibility.

8. Bibliography

1: Gameloft, http://www.gameloft.com
2: iOS, https://developer.apple.com/ios/
3: Android, http://www.android.com
4: Tizen, http://www.tizen.org
5: CMake, https://cmake.org
6: qmake, http://doc.qt.io/qt-5/qmake-manual.html
7: Premake, http://premake.github.io
8: Conan package manager, https://www.conan.io
9: biicode, http://www.biicode.com
10: CPM, http://www.cpm.rocks
11: Pacm, http://sourcey.com/pacm
12: CVM, https://github.com/Offirmo/cvm
13: The Lua programming language, http://www.lua.org
14: Asana, http://asana.com
15: Subversion, https://subversion.apache.org/
16: Subversion's externals definition, http://svnbook.red-bean.com/en/1.2/svn.advanced.externals.html
17: Boost C++ Libraries, http://www.boost.org/
18: So you want to write a package manager,
19: Nix Package manager, https://nixos.org/nix/
20: RPM, http://www.rpm.org/
21: Apt, https://wiki.debian.org/Apt
22: Fink, http://www.finkproject.org/
23: Steam, http://store.steampowered.com/
24: pkgsrc, http://www.pkgsrc.org/
25: Pacman, https://www.archlinux.org/pacman/
26: pkgutil, http://pkgutil.net/
27: Portage, https://wiki.gentoo.org/wiki/Project:Portage
28: MacPorts, http://www.macports.org/
29: Homebrew, http://brew.sh/
30: iOS App Store, https://www.apple.com/appstore
31: Chocolatey, http://chocolatey.org/
32: Google Play, https://play.google.com/
33: Zero install, http://0install.net/
34: Windows Store, http://apps.microsoft.com/
35: Mac App Store, http://www.apple.com/es/osx/apps/app-store/
36: FreeBSD Ports, http://www.freebsd.org/ports/
37: Nintendo eShop for Wii U, http://www.nintendo.com/wiiu/downloads/
38: Nintendo eShop for 3DS, http://www.nintendo.com/3ds/downloads/
39: Xbox Games Store, https://store.xbox.com/
40: PlayStation Store, https://store.playstation.com/
41: Go, https://golang.org/
42: C#, https://msdn.microsoft.com/en-us/library/kx37x362.aspx
43: pip, http://www.pip-installer.org/
44: PHP, http://php.net/
45: Perl Package Manager, https://metacpan.org/pod/PPM
46: PAR: Perl Archive Toolkit, http://par.perl.org/

47: npm, https://www.npmjs.com/
48: Ruby, http://www.ruby-lang.org/
49: PyPI: Python Package Index, https://pypi.python.org/pypi
50: PECL: PHP Extension Community Repository, http://pecl.php.net/
51: PEAR: PHP Extension and Application Repository, http://pear.php.net/
52: cpan, https://metacpan.org/pod/distribution/CPAN/scripts/cpan
53: Comprehensive Perl Archive Network (CPAN), http://www.cpan.org/
54: The Perl programming language, https://www.perl.org/
55: Java, https://www.java.com
56: Conda, http://conda.pydata.org/docs/
57: Swift, https://swift.org/
58: Maven, https://maven.apache.org/
59: CABAL: Common Architecture for Building Applications and Libraries, https://www.haskell.org/cabal/
60: Glide, https://glide.sh/
61: godep, https://github.com/tools/godep
62: NuGet, https://www.nuget.org/
63: Python, https://www.python.org/
64: LuaDist, http://luadist.org/
65: Hackage, https://hackage.haskell.org/
66: Haskell Language, https://www.haskell.org/
67: Go Tool, https://golang.org/cmd/go/
68: C / C++ reference, cppreference.com
69: RubyGems, https://rubygems.org/
70: Buildout, http://www.buildout.org/
71: Composer, https://getcomposer.org/
72: LuaRocks, https://luarocks.org/
73: Packagist, https://packagist.org/
74: PPM Index, http://code.activestate.com/ppm/
75: CocoaPods, https://cocoapods.org/
76: QuickLisp, https://www.quicklisp.org
77: JavaScript, https://en.wikipedia.org/wiki/JavaScript
78: Objective-C, https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html
79: Go packages, https://golang.org/pkg/
80: Common Lisp, https://common-lisp.net/
81: Apache Ivy, http://ant.apache.org/ivy/
82: cpanm, https://metacpan.org/pod/distribution/App-cpanminus/bin/cpanm
83: Maven Central, maven.org
84: Node.js, https://nodejs.org
85: Swift Package Manager, https://github.com/apple/swift-package-manager
86: cpanp, https://metacpan.org/pod/distribution/CPANPLUS/bin/cpanp
87: Gradle, http://gradle.org/
88: local::lib, https://metacpan.org/pod/local::lib
89: Bazel, http://bazel.io/
90: CoApp, http://coapp.org/
91: Make, https://en.wikipedia.org/wiki/Make_(software)
92: Xcode, https://developer.apple.com/xcode/
93: MSBuild, https://msdn.microsoft.com/en-us/library/dd393574.aspx

94: Apache Ant, http://ant.apache.org/
95: Jenkins, https://jenkins.io/
96: Git, https://git-scm.com/
97: GitLab, https://gitlab.com/
98: Microsoft Visual Studio, https://www.visualstudio.com/
99: Babelua VS extension, https://babelua.codeplex.com/
100: Vim, http://www.vim.org/
101: Notepad++, https://notepad-plus-plus.org/
102: GCC, the GNU Compiler Collection, https://gcc.gnu.org/
103: Decoda, http://unknownworlds.com/decoda/
104: Lua for Windows, https://github.com/rjpcomputing/luaforwindows
105: Vagrant, https://www.vagrantup.com/
106: Oracle VirtualBox, https://www.virtualbox.org/
107: Docker, https://www.docker.com/
108: Atlassian Jira, https://www.atlassian.com/software/jira
109: Atlassian Confluence, https://www.atlassian.com/software/confluence
110: LibreOffice, https://www.libreoffice.org/
111: yEd Graph Editor, https://www.yworks.com/products/yed
112: Microsoft Project, https://products.office.com/project/
113: MSBuild bugs are solved as "Won't fix", https://connect.microsoft.com/VisualStudio/feedback/details/1023369/msbuild-doesnt-parse-environment-variables-in-sln-files
114: NVIDIA CodeWorks for Android, https://developer.nvidia.com/codeworks-android
115: Semantic Versioning, http://semver.org/
116: Lua-users wiki, http://lua-users.org/wiki/
117: Penlight Lua Libraries, http://stevedonovan.github.io/Penlight
118: Lua's require function reference, https://www.lua.org/manual/5.1/manual.html#pdf-require
119: LuaFileSystem, http://keplerproject.github.io/luafilesystem/
120: lua-compat-5.3, https://github.com/keplerproject/lua-compat-5.3
121: tableshape, https://github.com/leafo/tableshape
122: typecheck, Gradual type checking for Lua, https://github.com/gvvaughan/typecheck
123: ansicolors.lua, https://github.com/kikito/ansicolors.lua
124: Jeffrey Friedl's Simple JSON Encode/Decode in pure Lua, http://regex.info/blog/lua/json
125: Telescope, https://github.com/norman/telescope
126: MoonScript, http://moonscript.org/
127: Typed Lua, https://github.com/andremm/typedlua
128: Metalua, http://metalua.luaforge.net/
129: API Diff, http://www.apidiff.com/en/
130: busted, elegant Lua unit testing, http://olivinelabs.com/busted/

8. Bibliography

PART 3. ANNEXES

9. Annex I: User manual

9.1. Install

9.1.1. Download

You can find the latest release and its download link in the Releases page.

9.1.2. Install

The GL-Get binary is a standalone command line application that can be used from any folder without the need to run a specific install process.

We recommend putting it in a folder visible to the system search path (using your operating system's PATH environment variable) for ease of use.

9.1.3. Basic configuration

GL-Get has an internal default configuration that should work out of the box on most cases. In case you need to change any setting, the procedure is described in the Configuration section.

You'll probably have to setup your SVN credentials in a config file. For this, create a file called "gl-get_config.lua" in the home folder of your user (usually c:\users\username) with content similar to this:

```
return
{
  vcs =
  {
        repo =
        {
              {
                    type = "svn",
                    location = "https://server_root/*",
                    username = "your_username",
                    password = "your_password"
              }
        }
  }
}
```

GL-Get uses an environment variable in order to refer to the shared folder. Decide where you want this folder to be (shared packages like Boost will be stored there, so it should have plenty of free space) and set the GL_GET_SHARED_FOLDER environment variable to point to this path. A sensible value can be something like "C:\GL-Get-packages" or "/var/db/GL-Get", depending on your operating system.

9.2. Configuration

WARNING: The config files are in fact Lua scripts, so keep in mind you'll have to escape any "\" in the specified paths using "\\".

The configuration files can be used to set a number of global settings. It mainly involves global settings to be used across the system.

These settings can be overridden by stacking definitions in several configuration files.

The "gl-get_config.lua" file will be looked for in the following locations, from more general to more concrete:

- In the home folder of the user
- In the folder of the binary (just on Windows)
- In the Workspace's .gl-get directory
- In the current working directory

It has the following format:

```
return
{
  curl =
  {
    path = "curl"
  },
  defaults =
  {
    abi_map =
    {
      Debug = "ds-er",
      Release = "os-",
      Gold = "ost-"
    }
  },
  manifest =
  {
    dirname = "manifest",
    filename = "_manifest.lua"
  },
  new_project =
  {
    configurations = { "Debug", "Release" },
    platforms = { "Windows:x86,x86_64" }
  },
  premake =
  {
    embedded_version = "5.0.0-alpha7-gl",
    path = "premake5.exe"
  },
  shared_folder =
  {
    environment_variable = "GL_GET_SHARED_FOLDER",
    --path = ""
  },
  svn =
  {
    path = "svn"
  },
  vcs =
  {
    default_type = "svn",
    repo =
    {
      {
        type = "svn",
        location = "https://default_server_location/"
      }
    }
  },
  version_constraints =
  {
    --["GL-Get-premake-utils"] = "0.0.1"
  },
  workspace =
  {
    tmp_dirname = ".gl-get"
  },
  default_build =
  {
    server =
    {
      url = "default_server_location",
      user = "default_user",
      token = "default_token",
    }
  }
}
```

9.2. Configuration

Settings are grouped in several sets:

- Tools:
  - **svn**: Subversion configuration
    - **path**: the path to the svn binary.
  - **curl**: cURL configuration
    - **path**: the path to the curl binary.
  - **premake**: Premake configuration
    - **path**: the path to the Premake binary.
    - **embedded_version**: the version number of the embedded Premake.
- Internal paths:
  - **manifest**
    - **dirname**: location inside the package where the manifest file is located.
    - **filename**: name of the manifest files.
  - **workspace**: this is the folder where the main package resides.
    - **tmp_dirname**: name of the temporary folder where the workspace information is cached.
  - **shared_folder**: this is the folder where the dependencies that are installed as common for all projects will go.
    - **environment_variable**: the name of the environment variable where the shared folder location will be taken from.
    - **path**: the actual location of the shared folder. It's automatically filled by the contents of the environment variable.
- Commands defaults:
  - **new_project**
    - **configurations**: list of configurations used to fill the newly created projects.
    - **platforms**: list of platforms used to fill the newly created projects.
  - **default_build**: settings regarding build configuration, both client and server side (Jenkins).
    - **server**: specifies a Jenkins server
      - **url**: the URL of the Jenkins host
      - **user**: the Jenkins user to operate on behalf of

        - **token**: the server token for the Jenkins server
    - **defaults**
        - **abi_map**: the default map of ABI flags for each configuration to use when building packages that don’t have it configured themselves.
- **version_constraints**: an array of versions associated to specific packages. Those versions will be used regardless of what version is requested in the dependency tree.
- **vcs**: configuration of the version control systems
    - **default_type**: the VCS type to use for the VCS configurations that don't specify an explicit type.
    - **repo**: an array of VCS configurations that will be used as matches to expand concrete VCS configurations. The complete documentation is available in the next section.

Each of the configuration files can contain any subset of these settings.

9.3. VCS configuration

A VCS configuration is a table that specifies several parameters. Most of the parameters are specific to the concrete VCS type. "type" is the only common parameter, which specifies the concrete VCS type to use, represented as a string. The available types at the moment of writing are "fs" and "svn", with "git" in the way. Each VCS type defines its valid parameters.

9.3.1. VCS types

fs

This is basically a VCS mock used in tests, backed by the filesystem. It has a single parameter:

- **location**: The location in the filesystem that will be used as the root for the repository.

Here’s a FS VCS configuration example:

```
{
  type = "fs",
  location = "c:/tmp/my_repo"
}
```

svn

Subversion. The valid parameters are:

- **location**: The URL to use as the root of the repository.
- **username**: The user name to use in the credentials to access the repository.
- **password**: The password corresponding to the specified user name to use in the credentials to access the repository.

9.3. VCS configuration

Here's a Svn VCS configuration example:

```
{
  type = "svn",
  location = "http://svn.example.com/repo",
  username = "my_username@example.com",
  password = "my_sikrit"
}
```

git

This is still WIP. The valid parameters are:

- **location**: The URL to use as the root of the repository.

Here's a Git VCS configuration example:

```
{
  type = "git",
  location = "http://git.example.com/repo"
}
```

9.3.2. Tips & tricks

Implicit VCS type

If the type of a VCS is skipped, the default is used. The default can be specified in the global config's *vcs.default_type*. So, for example, in the context of the following global configuration:

```
{
  vcs =
  {
    default_type = "svn"
  }
}
```

the following VCS configuration:

```
{
  location = "http://git.example.com/repo"
}
```

would result in the following effective VCS configuration:

```
{
  type = "svn",
  location = "http://git.example.com/repo"
}
```

Common VCS configurations

Sometimes you may want to share some parameters among several VCS configurations (like base packages locations or login credentials). This can be achieved using common VCS configurations, which can be configured in the global config's vcs array. Each entry is a VCS configuration itself, but the way they're used depends on some parameters:

Base packages locations

Most packages will be located on common repositories. These common locations can be declared as standard VCS configurations. These configurations can optionally have an associated name so they can easily be referenced from other places, like in package dependencies. Here's an example:

```
{
  vcs =
  {
    stable = {
      type = "svn",
      location = "http://svn.example.com/stable"
    },
    incubator = {
      type = "svn",
      location = "http://svn.example.com/incubator"
    }
    mine = {
      type = "fs",
      location = "c:/my_packages"
    }
  }
}
```

NOTE: Currently these base locations will only use the location field. Extra parameters will be ignored.

Configuration extenders

When some arguments are common to many VCS configurations, we can create a common VCS configuration that will be used as a pattern, using the location to match the concrete VCS configurations.

When specifying a concrete VCS configuration, it will be matched with the common configurations, first using the VCS type, and then using a VCS specific criteria, commonly matching the location, which can contain pattern globs (* and ?). In case several common configurations match a given VCS configuration, the selected will be the one with the "longer match" (longer location parameter).

Once a match is found, all the parameters in the common configuration missing in the concrete VCS configuration are copied to it, expanding the concrete configuration.

As an example, in the context of the following global configuration:

```
{
  vcs =
  {
    {
      type = "svn",
      location = "http://svn.example.com/repos/*",
      username = "repos_username",
      password = "repos_password"
    },
    {
      type = "svn",
      location = "http://*example.com/*",
      username = "global_username",
      password = "global_password"
    }
  }
}
```

the following VCS configuration:

```
{
  type = "svn",
  location = "http://svn.example.com/repos/application"
}
```

would result in the following effective VCS configuration:

```
{
  type = "svn",
  location = "http://svn.example.com/repos/application",
  username = "repos_username",
  password = "repos_password"
}
```

because it matched both common configurations, but the longest match was the first one.

This other VCS configuration:

```
{
  type = "svn",
  location = "http://example.com/libs/library",
  username = "custom_user"
}
```

would result in the following effective VCS configuration:

```
{
  type = "svn",
  location = "http://example.com/libs/library",
  username = "custom_user",
  password = "global_password"
}
```

because it just matched the second common configuration.

9.4. Actions

The current section will talk about the concepts required in order to use each of the actions, but it will not discuss the particular command line arguments. GL-Get is auto-documented, which means that the application itself can show help about each of the actions and their valid options. As a reference, chapter 10 gathers the current help messages as shown by the application.

9.4.1. Creating a package

It creates a new package, which can be used to start working on a new project.

By default it will create a sub-directory of the current working directory, named after the package, but a destination path can be specified.

It needs the name of the package and the information of the main project. When used without arguments, it will run a wizard that will ask for the missing information.

It can also optionally upload the new VCS repository.

9.4.2. Checking out

It checks out a package from the repository, together with its dependencies. There are several ways to specify the package to check out:

- With the package name and optionally the repository name.
- With the package repository URL.
- With the final URL that points to the concrete branch or tag.

When the final URL isn't given, some more information must be specified, like the package version, a branch or a tag.

A checkout mode and a profile can be specified if desired.

By default it will be checked out into a sub-directory of the current working directory, named after the package, but a destination path can be specified.

9.4.3. Updating

It can be used to download the latest changes from the package repository and its dependencies.

By default it will try to locate the workspace in the current working directory, but a destination path can be specified.

9.4.4. Generating the project files

It generates the project files for the current package. By default it also generates the project files of its dependencies, but they can be skipped.

By default it will try to locate the workspace in the current working directory, but a destination path can be specified.

It generates the project files for a specified toolset. If a toolset can target more than one platform, the target platform also has to be specified. In case a platform has a single toolset, the platform can also be specified without the toolset as an alternative. If nothing is specified, the current host platform will be used as a default.

If the toolset can only generate a single architecture at a time, the architecture will also have to be specified.

9.4.5. Building the projects

It builds all the projects in the package.

By default it will try to locate the workspace in the current working directory, but a destination path can be specified.

A toolset and a platform have to be specified, but one of them may be skipped in case it can be deduced from the other one, like for the generation action.

At least one or more architectures and configurations have to be specified. Optionally it can build all the combinations.

9.4.6. Running the tests

It runs the test projects of the package and checks for their result.

It uses the same options as the build action, in order to select which binaries have to be run.

9.4.7. Adding a project

It creates a new project on an existing package.

By default it will try to locate the workspace in the current working directory, but a destination path can be specified.

The name, type and the container project group of the new project are required. A UUID can be specified, but a new one is automatically generated otherwise.

9.4.8. Adding a dependency

It can be used to add a new dependency to an existing package.

By default it will try to locate the workspace in the current working directory, but a destination path can be specified.

Dependencies have two main groups of options:

- Where the dependency is added:
  - Directly into a project group: at least the project group has to be specified.
  - Into a project: both the project group and the project name are required.
  - If the new dependency references another package, a folder name has to be specified.
- Target of the dependency:
  - When it's an internal dependency: a project group is mandatory, and a project name is optional (not needed when we want to target a project group).
  - When the target is external, we have to specify the name of the target package and the related options required to access it: its repository name, the version, branch, tag, or the URL in case it's a legacy package. It can also optionally target the latest available version.

Optionally the dependency can be marked as being forced to be shared or not.

9.4.9. Validating the package

The validate action can be used to make sure everything's right in a package.

By default it will try to locate the workspace in the current working directory, but a destination path can be specified. It can also be given the same options as to the checkout action, and it will perform a temporary checkout in order to run all the checks.

Once it has a package workspace, it will generate the project files, it will build the projects, and it will run all the test projects, in addition to running some extra package validation checks.

For this it will need the same options as the build and test actions (the toolset, platform, architecture and configuration).

10. Annex II: Command line options reference

This chapter collects the command line documentation, as shown by the tool itself, with the available actions and their options. Some of the default values or valid values shown may change depending on the host system or its configuration.

10.1. Global options

This section shows the global options and the main available actions. All the global options can be specified with all the actions:

```
Usage: GL-Get.exe <Global options> <ACTION> [action-options]

GL-Get is an apt-get like dependency manager, capable of bootstraping, configuring
and building projects with a command line instruction.

Global options:

General options:
  -h [ --help ]                   Shows this help message
  --common-folder arg (=c:\glget) This is the folder where the dependencies
                                  that are installed as common for all projects
                                  will go
  --no-color                      Disables colored output

Utilities paths:
  --svn-path arg (=svn)                 Path of the Subversion executable
  --curl-path arg (=..\..\tools\curl\curl.exe)
                                        Path of the curl executable

Advanced options:
  --tmp-path arg (=.gl-get) Path of the temporary working directory
  -d [ --debug ] [=arg(=1)] Specifies the extra debug messages level
  --package-path arg        Adds the specified path to the lua package.path
  --premake-scripts arg     Specifies the location of Premake's scripts
  --commands-path arg       Path for looking for custom commands

Available actions:
     add-dependency  Add a dependency to a project
        add-project  Add a project to a package
              build  Builds a GL-Get project
           checkout  Checks out a versioned package
             create  Creates the a new package
           generate  Generates a GL-Get project
              glget  Helper commands for GL-Get developers
               info  Gives information about the current project_instance.
        show-config  Show GL-Get configuration
  show-dependencies  Shows the package dependencies
       show-version  Show version information for GL-Get
               test  Tests a GL-Get project
             update  Updates a GL-Get working copy
           validate  Verifies a GL-Get working copy
```

10.2. add-dependency action

```
Usage: GL-Get.exe <Global options> add-dependency <Add-dependency options>

Use this command to add a dependency to a GL-Get project.

Add-dependency options:

Parent project options:
  --destination arg     The destination path where the action will take place
  --project-group arg   The project group to add the project to ("" means the
                        main project group)
  --project arg         The project to add the project to ("" means the main
                        project)
  -f [ --folder ] arg   The folder to add the dependency to
  -s [ --shared ]       Add as shared dependency

Subversion options:
  -l [ --location ] arg The URL of the repository root
  --username arg        The svn user to use for authenticating with the server
  --password arg        The svn password matching the user specified in
                        svn.username

New dependency:
  -p [ --package ] arg  The name of the package
  -v [ --version ] arg  The package version
  --repo arg            The name of the repository to look for the package
  -t [ --tag ] arg      The VCS tag
  -b [ --branch ] arg   The VCS branch ("" means the main branch)
  -r [ --revision ] arg The VCS revision
  --vcs arg (=svn)      The VCS type for the package. The valid values are:
                        svn, fs
  --last-version        Use the last available version
```

10.3. add-project action

```
Usage: GL-Get.exe <Global options> add-project <Add-project options>

Use this command to add a project to a GL-Get package.

Add project options:
  --destination arg     The destination path where the action will take place
  --project arg         Project name
  --project-group arg   The project group to add the project to
  -t [ --type ] arg     The type of the project
  --uuid arg            The UUID of the project
```

10.4. build action

```
Usage: GL-Get.exe <Global options> build <Build options>

Use this command in order to verify if a current working copy is ready to be commited.

Build options:
  --destination arg          The destination path where the action will take
                             place
  -t [ --toolset ] arg       The target toolset. The valid toolsets for the
                             Windows platform are: VS2012, VS2013, VS2015.
  -l [ --platform ] arg      The target platform. The valid platforms are:
                             Android, iOS, Linux, MacOSX, tvOS, Windows,
                             Windows10.
  -a [ --architecture ] arg  The target architecture. The valid platforms for
                             the Windows platform are: X86, X86_64.
  -c [ --configuration ] arg The target configuration.
  --all                      Targets all architectures and configurations.
  --all-architectures        Targets all architectures.
  --all-configurations       Targets all configurations.
  --cleanup                  Erase the temporal folder after verify.
  -r [ --remote-url ] arg    Remote URL of the build server.
```

10.5. checkout action

```
Usage: GL-Get.exe <Global options> checkout <Checkout options>

Use this command to perform a checkout of a GL-Get package.

Checkout options:

Package to checkout:
  --destination arg           The destination path where the action will take
                              place
  -m [ --mode ] arg (=source) The checkout mode. The valid values are: source,
                              binary
  --profile arg               The package profile. The valid values are: devel,
                              main, full
  -p [ --package ] arg        The name of the package
  -v [ --version ] arg        The package version
  --repo arg                  The name of the repository to look for the
                              package
  -t [ --tag ] arg            The VCS tag
  -b [ --branch ] arg         The VCS branch ("" means the main branch)
  -r [ --revision ] arg       The VCS revision
  --vcs arg (=svn)            The VCS type for the package. The valid values
                              are: svn, fs

Subversion options:
  -l [ --location ] arg The URL of the repository root
  --username arg        The svn user to use for authenticating with the server
  --password arg        The svn password matching the user specified in
                        svn.username

Optional dependencies:
  --with-optionals      Install all the optional dependencies
  --with arg            List of optional dependencies to install
```

10.6. create action

```
Usage: GL-Get.exe <Global options> create <Create options>

It creates a new package by filling its manifest file and optionally creating its
VCS repository.

Create options:

New package options:
  --destination arg     The destination path where the action will take place
  -p [ --package ] arg  Package name
  -v [ --version ] arg  Initial package version
  --skip-vcs            Skip creating the VCS repository
  --vcs arg             The VCS type for the package. The valid values are:
                        svn, fs

Main project options:
  -t [ --type ] arg     The type of the project
  --uuid arg            The UUID of the project
```

10.7. generate action

```
Usage: GL-Get.exe <Global options> generate <Generate options>

Use this command in order to generate the project files of the package.

Generate options:
  --destination arg         The destination path where the action will take
                            place
  -t [ --toolset ] arg      The target toolset. The valid toolsets for the
                            Windows platform are: VS2012, VS2013, VS2015.
  -l [ --platform ] arg     The target platform. The valid platforms are:
                            Android, iOS, Linux, MacOSX, tvOS, Windows,
                            Windows10.
  -a [ --architecture ] arg The target architecture. The valid platforms for
                            the Windows platform are: X86, X86_64.
  --skip-deps               Skip the dependencies, only generate the main
                            package
```

10.8. glget action

```
Usage: GL-Get.exe <Global options> glget <ACTION> [action-options]

Helper commands for GL-Get developers

Available actions:
  release  Releases a new version of GL-Get
```

10.8.1. release action

```
Usage: GL-Get.exe <Global options> glget release

Releases a new version of GL-Get
```

10.9. info action

```
Usage: GL-Get.exe <Global options> info <Info options>

Use this command to view information about the current project_instance.

Info options:
  --destination arg     The destination path where the action will take place
```

10.10. show-config action

```
Usage: GL-Get.exe <Global options> show-config

Use this command to show the configuration GL-Get will use.
```

10.11. show-dependencies action

```
Usage: GL-Get.exe <Global options> show-dependencies <Show-dependencies options>

Use this command to visualize the dependency tree of the main package of a workspace.

show-dependencies options:
  --destination arg     The destination path where the action will take place
```

10.12. show-version action

```
Usage: GL-Get.exe <Global options> show-version

Use this command to display version information for the GL-Get tool.
```

10.13. test action

```
Usage: GL-Get.exe <Global options> test <Test options>

Use this command in order to verify if a current working copy is ready to be commited.

Test options:
  --destination arg          The destination path where the action will take
                             place
  -t [ --toolset ] arg       The target toolset. The valid toolsets for the
                             Windows platform are: VS2012, VS2013, VS2015.
  -l [ --platform ] arg      The target platform. The valid platforms are:
                             Android, iOS, Linux, MacOSX, tvOS, Windows,
                             Windows10.
  -a [ --architecture ] arg  The target architecture. The valid platforms for
                             the Windows platform are: X86, X86_64.
  -c [ --configuration ] arg The target configuration.
  --all                      Targets all architectures and configurations.
  --all-architectures        Targets all architectures.
  --all-configurations       Targets all configurations.
```

10.14. update action

```
Usage: GL-Get.exe <Global options> update <Update options>

Use this command in order to update a GL-Get working copy previously checked out with
the checkout command

Update options:
  --destination arg     The destination path where the action will take place
```

10.15. validate action

```
Usage: GL-Get.exe <Global options> validate <Validate options>

Use this command in order to verify if a current working copy is ready to be commited.

Validate options:

Build information:
  --destination arg          The destination path where the action will take
                             place
  -l [ --platform ] arg      target platform to be verified.
  -a [ --architecture ] arg  target architecture to be verified.
  -o [ --toolset ] arg       target toolset.
  -c [ --configuration ] arg target configuration to be verified.
  --cleanup                  erase the temporal folder after verify.

Subversion options:
  -l [ --location ] arg The URL of the repository root
  --username arg        The svn user to use for authenticating with the server
  --password arg        The svn password matching the user specified in
                        svn.username

Validate svn information:
  -p [ --package ] arg  The name of the package
  -v [ --version ] arg  The package version
  --repo arg            The name of the repository to look for the package
  -t [ --tag ] arg      The VCS tag
  -b [ --branch ] arg   The VCS branch ("" means the main branch)
  -r [ --revision ] arg The VCS revision
  --vcs arg (=svn)      The VCS type for the package. The valid values are:
                        svn, fs
```