Design Patterns in Package Managers

Peter Fačko

April 23, 2023

Abstract

A standardized package manager is one of the most common features of modern programming languages and yet in C++, it is one of the most missed features by its users. In this article, we examine multiple existing package managers from which we derive a number of design patterns. In order to increase accessibility, these patterns are described using a structured pattern language.

1 Introduction

The 2022 annual survey of C++ developers done by the Standard C++ Foundation contained these two questions: "Which of these do you find frustrating about C++ development?" and "How do you manage your C++ 1st and 3rd party libraries?" [2]. The top answers were "Managing libraries my application depends on" and "The library source code is part of my build", respectively. Here is an example selection of languages with a standard or a de facto standard package manager: D, Go, JavaScript, .NET, Python, Ruby, and Rust. These two facts together show that 1) a package manager is a desired and common feature of any programming language and 2) C++ doesn't have a package manager and it is its most frustrating non-feature. In this paper, we examine multiple package managers mostly from the C++ environment and from them, we derive some of the most frequently occurring design patterns. We describe the patterns using a pattern language similar to the one used in the article "A Pattern Language of an Exploratory Programming Workspace" [3].

We'll use the following pattern language: First, we introduce the problem context for the pattern. Then we describe the problem solution, which is the description of the pattern. These are the parts most useful to the majority of readers. Next comes a section with pattern drawbacks, as patterns don't typically come without a price. Since these patterns are derived from a specific package manager analysis, we mention the specific examples of pattern occurrence in real software next. Lastly, we reference related patterns, as the discovered patterns can be grouped by their problem scope into a more comprehensible structure.

The analyzed package managers are Conan, vcpkg, nix, tipi.build, build2, and npm.

2 Design patterns

2.1 Package context

2.1.1 Problem

Managed packages are always inherently relative to some context. In the typical case of an operating system package manager, the context is the whole system. The fact that in such cases the context is very broad and also implicit, makes it very difficult for the manager and the user to take advantage of the existence of package contexts.

Consider a situation where there are two projects, project A and B, developed on the same machine with package management done by a typical OS package manager. Project A requires package P with version V1 and project requires P version V2. Suppose that the simultaneous presence of versions V1 and V2 would lead to the diamond problem. Since the context for the OS manager is the whole system, the manager cannot allow having both versions of package P installed.

When upgrading, removing, or changing the installed packages in any way, there is a small possibility that some binary from the package will run during the transformation process. Having this operation be non-atomic could lead to unexpected runtime behavior of the package.

2.1.2 Solution

To make the package context concept more useful, managers can make package contexts explicit in two ways. One is documentation, where the existence of package contexts is explicitly discussed as a property of the manager. Another is to implement package contexts as an entity in the manager's software architecture. Having an explicit entity could allow the user to create, configure, and reference contexts, which in turn enables some features which will be mentioned later. To address the broadness typical for managers not concerned with package contexts, the manager provides a way to specify the affected area for a package context or requires the consumer of a package to specify the context to which the consumption is relative.

A manager providing explicit contexts allows an easy solution to the situation of multiple projects requiring different package versions because each project can be relative to a different context. A way to not provide arbitrary contexts while still solving this problem is to provide project-local contexts, where each project created with the manager gets assigned a newly created context associated only with the new project.

Atomic upgrades, rollbacks, and similar features can be implemented using explicit package contexts. For each transformation, the manager can create a new context with the transformation applied in the new context and the old context left intact. When a way of atomically switching between the used context is applied, e.g. changing a symbolic link, the atomic transformation can be done by non-atomic transformation in the new context, which isn't yet used by anything and so doesn't affect anything, and then by atomically switching the context.

2.1.3 Drawbacks

As the concept of a package context is always present in the package manager design, there is no real drawback to introducing explicit contexts in terms of functionality.

However, since having the contexts explicit increases the complexity of a manager's design, there is a drawback of increased development time.

Also, when implemented explicitly, contexts are by their nature not optional in terms of the user's choice of utilized features. This can lead to a steeper learning curve for the user, which could also be considered a disadvantage over a more implicit solution.

2.1.4 Known uses

vcpkg and npm provide explicitly documented project-local contexts.

build2 provides explicit context creation with linking which allow the user to create a base context with common content used by multiple inheriting contexts.

nix provides user-local packages and atomic package transformations using package contexts and symbolic links.

2.2 The manager is a package

2.2.1 Problem

A package manager is a piece of software and as such it needs to be distributed to its end users. Software distribution is itself a problem worth exploring in great detail, but for purposes of this document let's assume that software distribution is hard and therefore practices that reduce its complexity can often be worth the development time.

As a piece of software, the manager also needs to be maintained. This mainly means being updated and installed or removed.

Software distribution is simultaneously one of the main features of package managers. However, the distribution capabilities provided by the manager can be quite limited, as it typically only needs to support distributing packages for the specific environment it is tailored to.

2.2.2 Solution

The solution to the problem of package manager distribution is to provide the manager as a package managed by the manager itself. This reduces the complexity of its distribution significantly since the

design of package distribution implemented by the manager is reused and reapplied for this purpose and all its guarantees and good properties are carried over.

As a byproduct, the manager can now be installed, updated, removed, and in general reliably maintained in a way that is well understood both by the users and the developers.

Having the manager provide itself as a package requires the manager to be implemented in such a way that it can handle its own installation. The installation could also possibly involve building which would put quite strong restrictions on the implementation of the manager.

A problem that arises with this pattern is the cyclical situation where a user wants to install the manager yet the manager is required for its installation. This must be solved by a bootstrap method, where the manager provides another way of distribution. This distribution can be very limited as it only needs to support installing the manager itself.

As a bonus, this pattern provides the manager with a good tutorial for the users, as the first required step of using the manager is to install it. The developers can then control the first experience of the user because they are in control of the manager and no other package is involved in this beginning stage yet.

2.2.3 Drawbacks

The main problem with this pattern is that the package manager must be implemented with the feature in mind. This might have small implications for some managers but for managers which for example distribute only from source in a certain language, this can put significant constraints on the implementation,

It also isn't clear that every manager would benefit from this pattern, as it is dependent on the package management features provided. Some managers might have such a feature set that this pattern introduces a large amount of complexity with very little benefit.

2.2.4 Known uses

npm provides itself as a package, although the usage is a bit more complicated and the typical usage requires an additional "meta" manager.

build2 provides itself and multiple other components of its toolchain as packages. By supporting build-time dependencies, the manager is also well integrated into its own ecosystem.

2.3 Default values

2.3.1 Problem

Package managers are complex pieces of software that require the user to input a great amount of configuration. Although the documentation for the options, parameters, and switches is typically provided, users can still find it difficult to understand the functionality of all the configuration parameters provided. Some parameters don't have to be specified, so if the user doesn't understand them they can leave them out. But with some parameters, the user must pick one option from many, which is a problem for the user if they don't understand the implications of individual values. The steep learning curve could deter users from using complex and feature-rich managers.

2.3.2 Solution

The solution is to use default values for most of the configuration. The manager can then expand its feature set indefinitely if it provides sensible default values for the newly introduced parameters.

The sensible choice for a value typically constitutes the value used by the majority of users or use cases. Another option for the default is to choose the safest option, which might not be the most common one but is the least probable to cause problems to the user.

2.3.3 Drawbacks

The default values pattern is best manifested in the situation when a user doesn't understand a certain option but doesn't have to bother with it because the manager uses a default value that has a very high chance of working correctly without the user even noticing. This advantage is also a

major drawback as the option is by design somewhat hidden from the user and thus possibly skewing the user's understanding of the manager. It is often the case that a configuration option is tightly associated with a specific feature. This means that the only place where a user might find out about a defaulted option and thus commonly also the feature associated with it is the documentation, which is often not consulted unless it is absolutely obvious and necessary. Default values make the needed consultation of documentation often necessary but not so obvious.

2.3.4 Known uses

Almost all software utilizes default values.

2.3.5 Related patterns

A more advanced and complex pattern in the area of default behavior is the Scanning pattern.

2.4 Scanning

2.4.1 Problem

The Default values pattern applies only to situations when a configuration parameter accepts a finite set of values or a number. If the type of the parameter is for example a string, it is often not possible for the developers to choose a sensible value that applies to most cases.

2.4.2 Solution

A good solution for some parameters with complicated types is to not provide a default value and throw an error when the user doesn't specify some value explicitly. However, a lot of configuration passed to the manager often repeats the same values that are part of the content of the package. These are for example the package name, version, binary name, and so on.

The manager uses a scanning tool that parses the package content and looks for the desired values. This might be an external tool in the case of source code or a parsing library for a known format such as JSON.

Package managers often come with their so-called manifest files which describe the package, but this pattern is more concerned with scanning for values in the content of a package that isn't there explicitly for consumption by the manager.

Sometimes the user wants to use a value for an option that is related to the value found inside the package content but is not the exact same. For this case, the manager can provide another parameter that would specify the transformation that ought to be done to the found value.

2.4.3 Drawbacks

Scanning is more complex and involved than just picking one default value, so it makes the manager harder to use.

The other difficulty is in the manager's implementation complexity, as scanning might require a lot of additional development. For example, parsing a directory structure is quite simple, but parsing code or arbitrary text files for default values is much more difficult.

2.4.4 Known uses

tipi.build uses scanning as part of its "build by convention" design.

2.4.5 Related patterns

Default values deals with a simpler case of default configuration.

2.5 Lockfiles

2.5.1 Problem

Many managers allow users to specify a dependency without specifying the version or by providing a version range. This means that the manager might have multiple versions of the package to choose from.

A very common behavior is to choose the newest available version. Although any backward-compatible version is supposed to work correctly, users sometimes wish to achieve reproducible builds. Reproducible builds improve debugging by disallowing a dependency to change the behavior of the project, which also ensures greater portability. They also help prevent the Dependency Confusion Attack where an attacker releases what to the manager looks like a benign package update which injects malicious code into its dependents.

2.5.2 Solution

A solution for reproducible builds is to use a so-called lockfile. A lockfile is a structure that contains information about the dependency tree of the built package. This information is complete, meaning that the whole dependency tree can be reconstructed using the lockfile.

It is common for managers to implement lockfiles as literal files which are passed as an argument when invoking the manager.

2.5.3 Drawbacks

An obvious drawback is that lockfiles require an additional structure in the project configuration which needs to be passed to the manager during the invocation and complicates the usage of the manager. The passing of the lockfile can be made implicit but that in turn only hides the added complexity.

A bigger problem with this pattern is that lockfiles are possibly a solution to a symptom of a bigger problem: a manager choosing the newest version of a package by default.

2.5.4 Known uses

Conan and npm offer lockfiles as one of their main features concerning versioning.

2.5.5 Related patterns

Baseline is a more robust solution to the problem of an updated package dependency. However, it comes with more constraints.

2.6 Baseline

2.6.1 Problem

One approach to package version specification is for the manager to allow arbitrary version ranges and pick the newest available version. Then a problem of unreproducible builds arises, which can be solved by lockfiles.

From another point of view, the possibility of a new version breaking its dependents is a problem. The existence of lockfiles can also be considered a problem, as it is a whole new mechanism that developers need to develop and users learn to use.

Another problem related to package versions is that when a newer version of a package can cause a dependency error requiring an additional fix, depending only on package versions when considering compatibility is not a reliable solution.

2.6.2 Solution

When a package manager allows specifying only left-bounded version ranges while always choosing the minimal version satisfying all dependencies, the problem of a new package dependency version breaking the project is solved [1].

Package managers often provide global package repositories. These can be utilized to address the problem of versioning unreliably ensuring compatibility. A way to ensure maximal compatibility

between packages in a registry is to have a CI system in the repository which triggers with any change and builds all affected packages (dependencies and dependents) and rejects any change to the repository which causes a build failure. For each package upgrade a new snapshot of the repository is registered. These snapshots are called baselines. The user then specifies dependencies to the manager as relative to some baseline and the manager resolves dependency versions to at least the value in the baseline. This way the manager provides a mechanism for ensuring package compatibility with the option to use newer, untested, versions of packages. Although the newer versions might still break compatibility, they are required to be specified by the user explicitly.

2.6.3 Drawbacks

Although the Lockfiles pattern arguably introduces unnecessary complexity to a manager's design, Baseline comes with its own complexity and is also more involved.

Moreover, baselines, in contrast to lockfiles, cannot be avoided by the user as they must stand at the base of dependency version specification in managers using them.

2.6.4 Known uses

From the analyzed packages only vcpkg offers baselines.

2.6.5 Related patterns

The Lockfiles pattern solves a similar problem in a more direct way.

2.7 Binary cache

2.7.1 Problem

Developers of package managers have an important design decision to make: whether to distribute source code or binary. These types of distribution are not mutually exclusive but each has its own incompatible advantages.

Distributing source is highly portable because each platform should be able to build binaries for itself. Next, it might result in faster code as additional optimizations specific to the target architecture could be applied. Lastly, when distributing source code there is no need to ensure binary compatibility as there is no binary distributed.

Binary distribution is on the other hand faster because it doesn't involve build time which is especially slow in C++. For the same reason, this type of distribution is in some ways simpler for the manager since there isn't a need to invoke the build.

2.7.2 Solution

To get the benefits of both building from source and directly distributing binaries, the manager can provide binary caching. When a user requests a package, the manager first looks up the package in the cache and only if it is missing there it will invoke the build from source.

Determining whether the cache contains some package is not trivial and requires a complex solution that ensures binary compatibility. However, the manager can always ignore the cache and build from the source as caching is only an optimization and building always yields the correct binary.

2.7.3 Drawbacks

This pattern comes with its own set of problems to solve but no strong drawback. Caches are an optimization and can be ignored.

2.7.4 Known uses

vcpkg, Conan, nix, and tipi.build all provide binary caching. The only manager building from source not supporting binary caching is build2, but it too has plans to support it in the future.

3 Conclusion

We described 7 design patterns that appear in some of the most used package managers in the C++ environment. This collection of patterns can help with designing future package managers for which there is an urgent need in C++ according to recent developer surveys.

References

[1] Russ Cox. *Minimal version selection*. URL: https://research.swtch.com/vgo-mvs.

[2] Standard C++ Foundation. *2022 Annual C++ Developer Survey "Lite"*. URL: https://isocpp.org/blog/2022/06/results-summary-2022-annual-cpp-developer-survey-lite.

[3] Marcel Taeumel et al. "A Pattern Language of an Exploratory Programming Workspace". In: *Design Thinking Research: Achieving Real Innovation*. Springer, 2022, pp. 111–145.