Package Ecosystem Plan

Rene Rivera – grafikrobot@gmail.com – P1177R1, 2018-11-15

Table of Contents

1. Abstract
2. Changes
    2.1. R1 (WIP, San Diego 2018)
3. Introduction
4. Terminology
5. Current Landscape
6. The Future
    6.1. Package Index
    6.2. Interfaces
7. Plan
    7.1. Compiler API
        7.1.1. Standard Options
    7.2. Build System API
    7.3. PDM API
        7.3.1. Package Dependency
        7.3.2. Packages Description
        7.3.3. Universal Package Identifier
    7.4. Package Index API
        7.4.1. Package Query
        7.4.2. Package List
    7.5. Package Index Authority
    7.6. Current Work
        7.6.1. Notes on C++ Package Management
        7.6.2. Implicit Module Partition Lookup
        7.6.3. A Module Mapper
        7.6.4. `std::compile`
        7.6.5. Let’s Talk About Package Specification
        7.6.6. `libman` A Dependency Manager ➔ Build System Bridge
        7.6.7. Canonical Project Structure
        7.6.8. The Pitchfork Layout (PFL)
8. Acknowledgements

Document number	ISO/IEC/JTC1/SC22/WG21/P1177R1
Date	2018-11-15
Reply-to	Rene Rivera, grafikrobot@gmail.com
Audience	Tooling (SG15)

1. Abstract

A plan for a cohesive ecosystem of package production and consumption among all C++ tools.

2. Changes

2.1. R1 (WIP, San Diego 2018)

Added "Current Work" section for apropos proposals: P1178, P1184, P1204, P1254, P1302, P1313, `libman`, and PFL.

3. Introduction

We’ve reached a point in C++ where our success has trapped us in a conundrum. The C++ Language and the standard library is a popular and high quality platform upon which to program for many domains. As such it is a popular vehicle for extensions for narrower and narrower domains. Which brings the unfortunate problem of making the management of those additions take more and more effort. The well known solution to this problem is to adopt package and dependency management as an extension to the built-in platform capabilities. Unfortunately for C++ the package and dependency management solutions to date are not interoperable nor interchangeable and hence cause confusion for users. As not only do they have to choose a compiler and environment, but now they also get to choose a package manager.

This paper intends to define a general structure of a package ecosystem wherein users can expect to consume and produce libraries with whatever C++ tools they choose in an interchangeable manner.

This paper **does not** propose to create a single standard build system, dependency manager, package manager, nor packages.

4. Terminology

Some of the terms used herein have historically varied meanings. In this paper here are what we intend when we refer to the terms.

Library

The group of code, compiled or as source, that you use directly. For example: libz, Boost MP11, and QtCore.

Package

An object that defines, and possibly implements, everything a user needs to use a library in your project. It can be a combination of some or all of: source for a library, pre-built binaries for a library, instructions for building the library, instructions for downloading binaries, enumeration of dependencies of the library.

Dependency Manager

Controls how the use of libraries in your project correlates to corresponding packages that provide those libraries to you project and how to arrange those package libraries to make them usable by your project.

Package Manager

Takes references to packages your project wants and delivers them such that the dependency manager can instruct your project to use those packages. This can include downloading from a non-local resource and building for the use case of your project.

Package and Dependency Manager

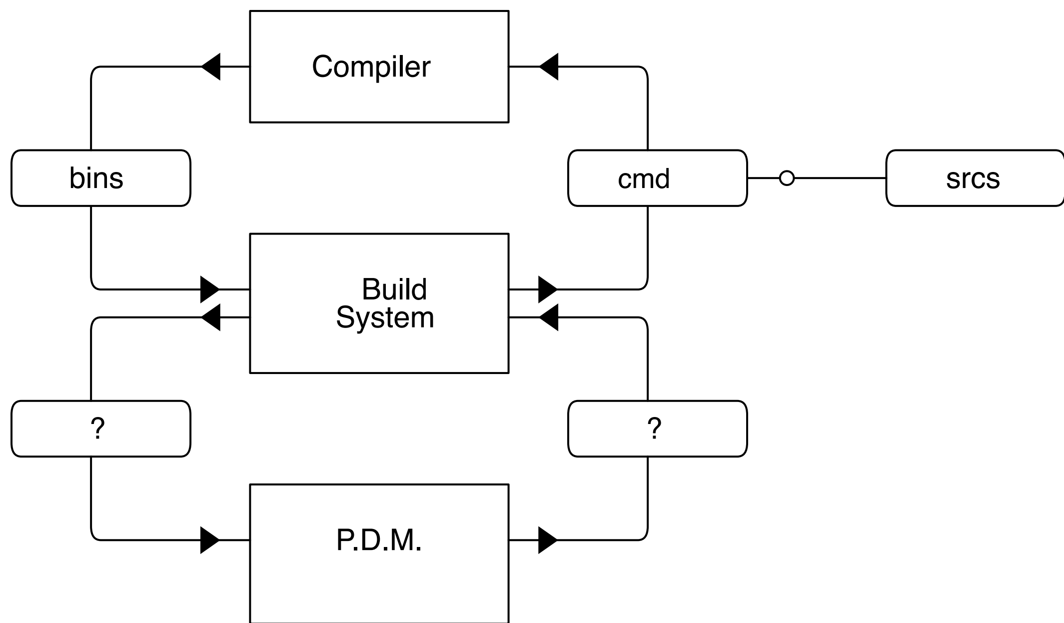
It is common, because of the close dependence between them, to combine both the dependency manager and the package manager. In this paper this is the form which we consider as it simplifies the discussion and is the form generally found in the wild.

5. Current Landscape

At this time we have three key components to our package ecosystem:

- Compiler
- Build System
- Package and Dependency Manager (PDM), and related packages

Below is a simplified view of the data flow between those components. As depending on the build system and PDM they may use additional data including source files.



Data	▢	Process	□
X Produces Y Y Accepts X	X ─▶── Y	X Uses Y	X ─o── Y

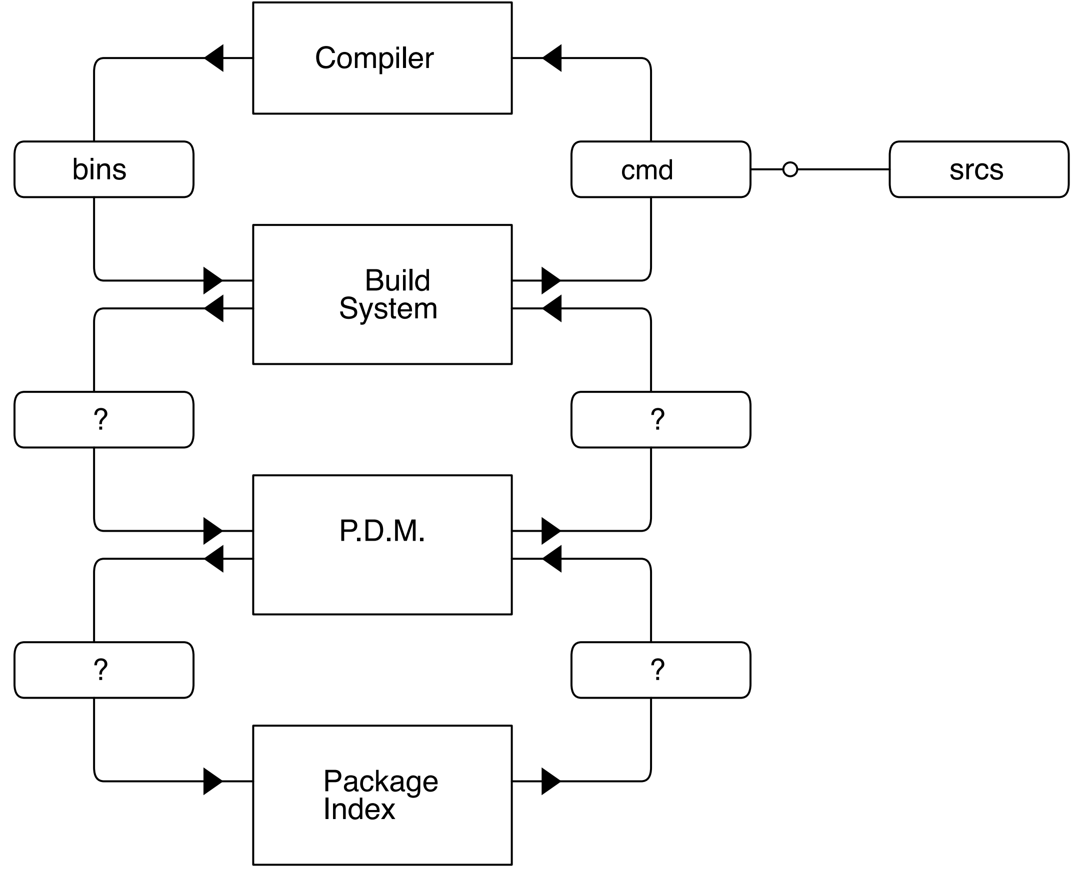
6. The Future

6.1. Package Index

One key problem with the current situation is that users consume packages from a restricted universe. When you decide to use PDM *A* you also decide to use the packages available **only** from *A*. Hence if there is a package *P* only available from PDM *B* you are back to some uncomfortable choices for your development.

That is not the only problem; There is also the issue that discovering what packages, from which PDMs, would involve going through each PDM and searching for what they have.

To resolve these we see the need for a global "Package Index". The index would contain data for each library in the C++ ecosystem. It would contain canonical information for each library such as: available packages (and the PDM of each), library name, description, license, and so on. What it would not contain would be the packages themselves. Having this would allow users to determine quickly where to get the one library they are interested in, hopefully with minimal effort.

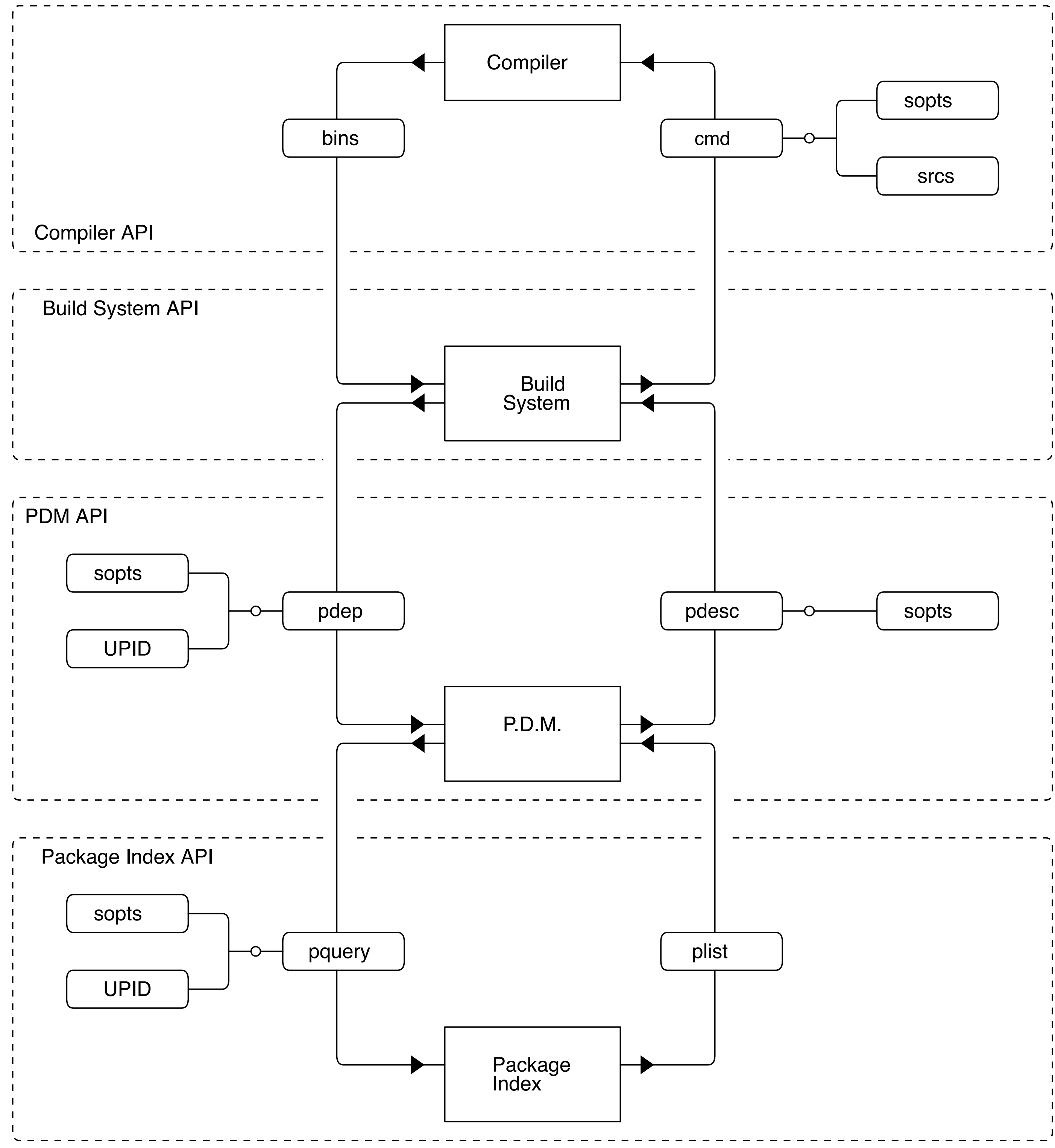


6.2. Interfaces

There is one glaring problem with the above future structure: Users are now not only dealing with how to interact with multiple compilers and build systems, but now would also have to deal with multiple PDMs. Thankfully, they might be spared having to deal with multiple package indices. We need to create standard interfaces to all our tools such that we gain the varied benefits from the common understanding they bring. What does it mean to create such standard interfaces? In terms of packages it means defining standards for:

- Invoking compilers
- Compiler responses
- Build system consumption and production of packages
- Communicating with a Package Index

We need to define all the data flow connections in our ecosystem in a standard and interchangeable manner:



In the diagram the following are used as shorthands:

`bins`	General output from compilers, might not be just binaries
`cmd`	Compiler commands
`srcs`	Language source we feed our tools with

`sopts`	Standard Options
`pdep`	Package Dependency
`pdesc`	Package Description
`UPID`	Universal Package Identifier
`pquery`	Package Query
`plist`	Package List

7. Plan

Note that this is a simplified view of the breadth of what such interfaces would need to specify. It is limited to the minimal interactions for using packages. Proposals for the individual components would be sufficiently detailed to handle the varied use cases of the tools ecosystem as a whole.

7.1. Compiler API

Goal

> Define standard interfaces to control and communicate with compilers to produce data required for build systems, PDMs, source editors, and so on.

Given that compilers are at the core of our data needs we would expect this API to expand over time to address the various needs of the tools that need it.

7.1.1. Standard Options

Currently build tools, and users, need to deal with a differing set of APIs to communicate with compilers. This makes it difficult to define consistent build descriptions not just in the build system but throught any tooling that needs reproducible builds.

In an ideal ecosystem all compilers would use a well known single options API that the whole ecosystem would use to interoperate between different compilers, build systems, PDMs, and other tools.

7.2. Build System API

Goal

> Define minimal standard interface to build and define software build requirements.

7.3. PDM API

Goal

> Define standard interfaces to control and communicate with PDMs to both consume and produce packages.

7.3.1. Package Dependency

Currently when using a PDM one uses a package reference particular to that PDM to indicate your dependency on a package. This makes using different PDMs difficult. It means that if you ever want to share your project with someone who uses a different PDM they need to rewrite all those package references, assuming equivalents even exist in that other PDM.

We need to define a single syntax to describe the packages we depend on. Having that would allow build systems to use PDMs interchangeably openning the entire collection of packages to users.

7.3.2. Packages Description

Like with the package dependency, we currently use different ways of describing the packages we need for any particular build of our software depending on the PDM we are using. We can't write a tool, like a build system, to communicate with PDMs one time. And like compilers we end up needing to write synthetic interfaces to each PDM on every tool.

Like having the common standard options, having common package description among PDMs allows us to write interchangeable tools on top of such an API. This API would combine the standard options and a package dependency to ask PDMs for the specific build variation we need to use. What happens behind the scenes to get you a package that matches your requirements is, as we are fond of saying, *implementation defined*. This API would: allow tool makers to "write once" to use PDMs, allow users to migrate from one PDM to another, it could even allow use of multiple PDMs simultaneously if such a need arrises.

7.3.3. Universal Package Identifier

To be able to support interchangeability between different PDMs we can't be tied to their individual package references. We need a single universal package identifier (UPID) to use in our projects that can refer to the components.

7.4. Package Index API

Goal

Define standard interfaces to query and publish package records for PDMs, and users.

7.4.1. Package Query

Having a package index is no good without a consistent way of asking for packages. A standard package query that tools can use to interrogate available packeges is just one of the many interface points we need in an index.

7.4.2. Package List

At minimum a package index needs to respond to queries with the set of packages available. The package list defines a standard response to that query.

7.5. Package Index Authority

Goal

Create a process for assigning a single authoritative source of package index information.

There is a key question that arises from having a package index.. Should we have one or many such indices? There are pros and cons for such a choice but there is one overwhelming concern.. Having multiple indices causes fragmentation and confusion for users. Assuming that there will be multiple parties interested in providing the index service we need to define who will be the authoritative source of this data. Hence we need to define a process that includes selection criteria, review, and official designation of who is the universal package index authority.

7.6. Current Work

7.6.1. Notes on C++ Package Management

Domain Interfaces

Authors Steve Downey <sdowney2@bloomberg.net>

Status Publish as P1254

Abstract Text interfaces should be defined to be used by package management, package build, and build systems to allow more effective C++ package management for users by allowing package management tools to drive the build of source packages without unhelpful help from the package's build system.

7.6.2. Implicit Module Partition Lookup

Domain Compiler API, Build System API

Authors Isabella Muerte https://twitter.com/slurpsmadrips, Richard Smith <richard@metafoo.co.uk>

Status Published as P1302

Abstract While we cannot enforce a project or directory layout for module interface units, we can encourage a general convention that developers, build systems, and compiler vendors can rely on for "quick paths" when developing software in C++.

7.6.3. A Module Mapper

Domain Compiler API

Authors Nathan Sidwell <nathan@acm.org>

Status Published as P1184

Abstract This paper describes an interface implemented as a serial protocol by which compilation tools may interrogate an entity encoding the mapping between module names (dotted identifier sequences or legacy header unit names), their interface source file and their Binary Module Interface. This interface allows a compiler to be agnostic about the mapping.

7.6.4. `std::compile`

Domain Compiler API, Standard Options

Authors Rene Rivera <grafikrobot@gmail.com>

Status Publish as P1178

Abstract This proposes to add an interface for transforming C++ source into usable executable programs. As such it aims to provide a common definition of compiler frontend tool arguments and options for transforming source code to translation units and linking those into executable programs.

7.6.5. Let's Talk About Package Specification

Domain PDM API, Packages Description

Authors Matthew Woehlke <mwoehlke.floss@gmail.com>

Status Publish as P1313

Abstract This paper explores the concept of a package specification — an important aspect of interaction between distinct software components — and recommends a possible direction for improvements in this area.

7.6.6. `libman` A Dependency Manager ➔ Build System Bridge

Domain PDM API, Packages Description

Authors Colby Pike <vectorofbool@gmail.com>

Status Pre-publication here https://api.csswg.org/bikeshed/?force=1&url=https://raw.githubusercontent.com/vector-of-bool/libman/develop/data/spec.bs (https://api.csswg.org/bikeshed/?force=1&url=https://raw.githubusercontent.com/vector-of-bool/libman/develop/data/spec.bs)

Abstract An exploration on a solution to giving build systems a way to consume packages presented to it by a dependency manager. This system is called `libman`, short for *Library Manifest*.

7.6.7. Canonical Project Structure

Domain Build System API, PDM API

Authors Boris Kolpackov <boris@codesynthesis.com>

Status Published as P1204

Abstract Source code layout and content guidelines for new C++ projects that would facilitate their packaging.

7.6.8. The Pitchfork Layout (PFL)

Domain Build System API, PDM API

Authors Colby Pike <vectorofbool@gmail.com>

Status Pre-publication here https://api.csswg.org/bikeshed/?force=1&url=https://raw.githubusercontent.com/vector-of-bool/pitchfork/develop/data/spec.bs (https://api.csswg.org/bikeshed/?force=1&url=https://raw.githubusercontent.com/vector-of-bool/pitchfork/develop/data/spec.bs)

Abstract PFL is a convention for laying out source, build, and resource files in a filesystem to aide in uniformity, tooling, understandability, and compartmentalization.

8. Acknowledgements

Thanks to CppCon 2017 for providing the environment for "arguments" about build systems, package managers, and dependency managers that created the impetus for this idea.

Thanks to Breno Rodrigues Guimarães, Steve Downey, and others in the CppLang Slack community who provided feedback to the draft version of this document.