# Common Lisp Project Manager

Eric Timmons
etimmons@mit.edu
CSAIL, Massachusetts Institute of Technology
Cambridge, MA, USA

## ABSTRACT

In this paper we describe and demonstrate the Common Lisp Project Manager (CLPM), a new addition to the Common Lisp dependency management ecosystem. CLPM provides a superset of features provided by the Quicklisp client, the current de facto project manager, while maintaining compatibility with both ASDF and the primary Quicklisp project distribution. These new features bring the Common Lisp dependency management experience closer to what can be achieved in other programming languages without mandating a particular development workflow/environment and without sacrificing Common Lisp's trademark interactive style of development.

## CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; *Software libraries and repositories*; *Application specific development environments.*

**ACM Reference Format:**
Eric Timmons. 2021. Common Lisp Project Manager. In *Proceedings of the 14th European Lisp Symposium (ELS'21).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.5281/zenodo.4716466

## 1 INTRODUCTION

One way to manage the complexity of software is to reuse code by packaging up a project and using it as a dependency in another. Unfortunately, this ends up introducing another level of complexity: you have to manage your dependencies and their versions as well! In the Common Lisp world, this management is usually performed by three interacting pieces: Another System Definition Facility (ASDF) [2] to ensure all dependencies are compiled and loaded in the right order and the version of each is sufficient, the Quicklisp client [7] for installing the dependencies locally and configuring ASDF to find them, and a Quicklisp distribution containing an index that lists available projects and metadata about them.

While effective and widely used, this combination of components is missing many features that are taken for granted in other programming language specific package management ecosystems. In this paper we describe the Common Lisp Project Manager (CLPM)[1], a potential replacement for the Quicklisp client in the Common Lisp dependency management ecosystem that provides many of these missing features.

We begin by first defining some terminology used throughout the paper. We then provide an overview of the tasks a dependency management solution must perform. Next we highlight several of the key features that CLPM implements, along with the overarching design philosophy. We then describe the high-level design of CLPM. Last, we provide information on how to obtain CLPM and several examples on how to use it.

[1]CLPM is in the process of being renamed from the Common Lisp Package Manager. While in most communities "package" typically refers to some installable unit of software, it unfortunately collides with the use of "package" to describe symbol namespaces in Common Lisp

## 2 TERMINOLOGY

In this section, we describe some of the terminology used throughout the rest of the paper. While the broad ideas are general, we do focus on Common Lisp and particularly on projects that use ASDF.

**Project** Primary development unit of code. Typically corresponds to the contents of a single version control system (VCS) repository, such as git.

**Release** A snapshot of a project. Typically identified using a version number, date, or VCS commit identifier. May contain multiple systems.

**System** Defined via ASDF's `defsystem`. Describes components (typically source code files) to be compiled/loaded, the version of the system, and dependencies on other systems.

**Dependency** A tuple $\langle f, s, v\rangle$. $f$ is a feature expression, $s$ is a system name, and $v$ is a version constraint. States that if $f$ is satisfied when evaluated against `*features*`, then a system with the name $s$ is required whose version satisfies $v$.[2]

**Source/Index** A collection of project and system metadata.

## 3 DEPENDENCY MANAGEMENT OVERVIEW

Broadly speaking, there are three tasks that any dependency management solution must perform: dependencies must be resolved, installed, and built. In this section, we explain each task, describe the Common Lisp tools involved in each (when using the current de facto workflow), and then provide further grounding by briefly explaining what tools are used for each task in selected other language ecosystems.

In the building phase, every dependency must be compiled and/or loaded, in the correct order. Additionally, there should be some feedback if dependencies are not met (for instance, system A needs version 2 of system B, but only version 1 is available). In current practice, ASDF performs this function. It uses system definitions to determine what dependencies any given system. Then it finds every dependent system, checks the version constraints, produces a plan that contains an ordered set of operations to perform, and executes it.

[2]Currently, ASDF only supports version constraints that specify a minimum version number.

However, ASDF has no support for installing dependencies; they must be placed on the file system or in the Lisp image by another tool. While it could be done manually, this installation phase is routinely performed by the Quicklisp client. The client fetches tarballs from the internet, unpacks them to the local file system, and configures ASDF so that it can find them.

Deciding which releases need to be installed is called dependency resolution. A highly manual approach to dependency resolution is to install a single release, try building it, see what's missing, install another release, and repeat. However, the process can be sped up using an index that specifies a list of known systems and their direct dependencies. The index can then be consulted recursively for each direct dependency (and their dependencies, and their dependencies, and so on) to produce a complete list of releases that need to be installed, before even attempting to build anything. This is the approach taken by Quicklisp. The client downloads a set of files containing this index from any distributions that it is configured to consult (such as the primary distribution, also named "quicklisp", hosted at https://beta.quicklisp.org/dist/quicklisp.txt).

While we have described each of these tasks as happening in distinct stages, in reality the tasks can be jumbled together. For instance, `ql:quickload` attempts to resolve and install all dependencies before invoking ASDF operations. However, it's possible that the Quicklisp distribution's index is missing some information (for example, if a local project is used that has no info in the index). Therefore, `ql:quickload` handles conditions that ASDF signals corresponding to missing dependencies by another round of resolution and installation.

## 3.1 Comparison to Other Languages

To further ground these concepts, we briefly describe the components responsible for each of these tasks in the Python and Ruby ecosystems.

In Python [18], building is mostly performed by the Python interpreter itself. When a package is `imported`, it is byte compiled if needed and then loaded into the interpreter. Pip [17] is the tool predominately used for dependency installation and resolution. During resolution, pip uses metadata gathered from the Python Package Index (PyPI) [10] and a `setup.py` file for packages not in the index.

In Ruby [16], building is performed by a combination of the gem tool [11] and the Ruby interpreter. Gem handles building native extensions (foreign libraries) while the Ruby interpreter loads `required` packages much like Python's interpreter. Dependency resolution and installation is typically handled with a combination of gem and bundler [4]. Gem is largely used for installing releases locally, while bundler is used for project specific installs. Both gem and bundler can consult indices such as the one hosted at https://rubygems.org/.

## 4 FEATURES AND PHILOSOPHY

CLPM has a set of features that, while commonly found in language specific package managers for other languages, have not seen wide adoption in the Common Lisp community. The most important of these include:

(1) the ability to download over HTTPS,
(2) the ability to reason over dependency version constraints during resolution,
(3) the ability to manage both global and project-local contexts,
(4) the ability to "lock" (or "pin", "freeze", etc.) dependencies to specific versions and replicate that setup exactly on another machine,
(5) the ability to compile to a standalone executable with a robust command line interface (CLI) for easy interfacing with shell scripts and other languages,
(6) and the ability to install development releases directly from a project's VCS.

While CLPM is not the first to implement most of these features for Common Lisp, we believe it is the first to do so in a complete package that also places minimal constraints on the workflow of the developer using it. For example, the quicklisp-https [14] project adds HTTPS support to the Quicklisp client, but it does so at the cost of requiring Dexador [8] (and all of its dependencies, including foreign SSL libraries) are loaded into the development image.

Another example is the qlot [9] project. It adds project-local contexts (but not global), locking, a CLI, and installing directly from VCS. However, it still requires that the Quicklisp client is installed and loaded. Until very recently its executable and CLI required Roswell [15]. To date its executable is not distributed and must be built locally.

To our knowledge, no other solution exists that attempts to include version constraints during resolution. To wit, Quicklisp's index format completely elides the version constraint in its dependency lists. So no project manager that uses only Quicklisp style indices for dependency resolution can ensure that a system's version constraints are satisfied.

As just alluded, the most important guiding principle of CLPM is that it should place minimal constraints on developer workflow. An example of this principle in action is illustrated by how CLPM is distributed. For Linux systems, a static executable is provided that runs on a wide variety of distributions with no dependencies on foreign libraries (some features, such as VCS integration, require other programs, such as git, to be installed). For MacOS systems, a binary is provided that requires only OS provided foreign libraries. And on Windows, CLPM is distributed using a standalone installer and again depends only on OS provided foreign libraries.

Perhaps the second most important principle is that CLPM should be highly configurable, yet provide a set of sane and safe defaults. A concrete example of this is shown by CLPM's behavior when installing or updating projects. By default, CLPM will describe the change about to be performed and require explicit user consent before making the modification. However, this behavior can be changed for developers that like to live on the edge or otherwise have complete trust in CLPM and the projects they are installing.

## 5 DESIGN

CLPM participates in both the installation and resolution phases of dependency management. It leaves building completely up to ASDF. In this section we discuss the design of CLPM and how it completes both of these tasks. First, we describe the overall architecture, including the three main CLPM components. Second, we describe some of the benefits the architecture provides and how

CLPM leverages them. Third, we describe where CLPM locates the metadata needed for dependency resolution. Last, we describe global and project-local contexts in CLPM.

## 5.1 Architecture

CLPM is split into two user facing components: the worker and the client, as well as one internal component: the groveler. The worker is a standalone executable that is responsible for all the heavy lifting. The worker interacts with the network to download releases and metadata, performs dependency resolution, unpacks archives, and manages contexts. The worker is implemented in Common Lisp and distributed both as a precompiled executable (static executable for Linux) and source code for those that want to compile it themselves. It exposes both a CLI interface and a REPL interface. The CLI interface allows for easy integration with tools such as shell scripts and continuous integration services. The REPL interface is used primarily by the CLPM client.

The client is a small system, written in portable Common Lisp, with ASDF/UIOP as its only dependency. The client is meant to be loaded into a Lisp image to facilitate the interactive management of dependencies and development of code. It does this by exposing a set of functions corresponding to the operations the worker can perform as well as integrating with ASDF's system search functions. Additionally, it has functionality to remove itself from an image if it is no longer required (such as when dumping an executable). In order to interact with the worker, the client spawns a new worker process, starts its REPL, and they communicate back and forth with S-expressions.

The last component is the groveler. Ideally users never interact directly with this component. Instead, it is used by the worker to gather the metadata needed for dependency resolution from systems that are not present in any index. For example, the groveler is used to extract dependencies from development versions of projects.

An aspect that makes extracting this metadata difficult is ASDF system definition files can contain arbitrary code and can require that certain dependencies be loaded before a system can even be correctly parsed. As such, it may not be possible to extract this metadata without running arbitrary code.

To address this, the groveler consists of a small set of portable Common Lisp functions that the worker loads into a fresh Common Lisp instance. Once loaded, the groveler and worker communicate via S-expressions, with the worker specifying which systems to load and extract information from and the groveler reporting back the information as well as any errors. If there is an error, the worker addresses it by recording any missing dependencies in the dependency resolution process, loading them into the groveler, and trying again. The worker keeps track of the systems loaded in the groveler and starts a new groveler process if needed (e.g., the groveler has v1 of system A loaded, but a dependency determined later in the resolution process requires v2).

Additionally, CLPM has experimental support to run the groveler in a sandbox. This sandbox has reduced permissions and cannot write to much of the file system. This sandbox is currently implemented using Firejail [6] on Linux, but we desire to add support for other OSes and sandbox solutions.

## 5.2 Dependencies and Non-portable Code in the Worker

One benefit of the worker/client split is that it enables the worker to freely leverage both dependencies (Common Lisp and foreign) and non-portable implementation features.

There are two benefits with respect to dependencies in the worker. First, the worker can reuse code without worrying about interfering with the code the user wishes to develop. This interference may manifest itself in many ways, including package nickname conflicts, version incompatibilities (CLPM needs version x of a system, but the user's code needs version y), or image size (e.g., the user's code uses few dependencies and they care about the final size of the image for application delivery).

Second, the worker can use foreign libraries that it is unlikely the user needs loaded for their development. A prime example of this is libssl, which CLPM uses on Linux and MacOS systems to provide HTTP support. A second-order benefit of this approach is that CLPM can statically link foreign libraries into the worker so that the user does not even need to have them installed locally (if they install a pre-compiled version of CLPM at least).

The ability for the worker to freely use dependencies has an additional knock-on effect: the development of CLPM can help improve the state of Common Lisp libraries at large. To date, the development of CLPM has resulted in at least ten issues and merge requests being reported to upstream maintainers, eight of which have been merged or otherwise addressed. Additionally, it has been the motivating factor behind several contributions to SBCL to improve musl libc support.

The worker routinely needs to perform operations that require functionality beyond the Common Lisp specification. For example: loading foreign libraries, interfacing with the OS to set file time stamps when unpacking archives, and network communication to download code. While portability libraries exist for many of these features, they are not perfect and do not necessarily extend support to all implementations [12]. The worker/client split allows us to choose a small number of target implementations and focus our testing and distribution efforts using only those implementations while not worrying about restricting what implementation the user uses to develop their code.

Currently, CLPM targets only SBCL [3] for worker support. This is due to SBCL's broad compatibility with OSes and CPU architectures. We look to extend worker support to at least one other implementation before CLPM reaches v1.0. The next implementation targeted by the worker will likely be ABCL [1] due to the ubiquity of the Java Virtual Machine.

## 5.3 Indices

Project indices are used to advertise projects and metadata about those projects, such as their released versions, what systems are available in each release, and the version number and dependencies of those systems. The most widely used project index in the Common Lisp community is the primary Quicklisp distribution. CLPM is tested against this index and has full support for interacting with it.

While there are other Quicklisp-like project indexes in the wild that work with the Quicklisp client, such as Ultralisp [5], CLPM may not work with all of them. This incompatibility is largely due to a lack of formal specification as to what constitutes a Quicklisp distribution, including what files are required and their contents. For instance, at the time of writing CLPM does not work with the Ultralisp distribution because Ultralisp does not publish a list of all its historical versions, unlike the primary Quicklisp distribution (see: https://beta.quicklisp.org/dist/quicklisp-versions.txt).

As discussed above, the Quicklisp distributions strip the version constraints from dependencies. However, CLPM actually supports reasoning over those constraints. This was the main motivation to develop and support Common Lisp Project Indices (CLPI). CLPI is part of the overarching CLPM project and seeks to fully document an index format as well as add features that CLPM can take advantage of that are missing from Quicklisp distributions. CLPI adds fields to record both a system's version (Quicklisp indices provide only the date at which the project snapshot was taken) and the version constraints of its dependencies.

A second major difference is that projects and systems are the top-level concepts in CLPI instead of distribution version. Additionally, CLPI defines the majority of its files to be append-only. These properties allow CLPM to be more efficient in terms of network usage as only the metadata for projects potentially needed in the context are transferred and metadata on new releases can be fetched incrementally over time.

## 5.4 Contexts

CLPM manages both global and project-local contexts. A context is defined as a set of project indices in use, a set of constraints describing the projects (and their versions) that should be available in the context, and a set of project releases that satisfy both the explicit constraints and the implicit constraints added by every project in the context (i.e., transitive dependencies).

Each global context is named by a string. CLPM provides tools that generate ASDF source registry configuration files so that you can add these global contexts to your default registry and have access to all projects installed in them without the need for CLPM at all (after installation, at least).

A project-local context (also known as a *bundle*) is named by a pathname that points to a file containing the first two elements of a context (the indices and constraints). After this context is installed, all of the context information is located in a file next to the file that names the context. The names of these files are typically `clpmfile` and `clpmfile.lock`. Both of these files are designed to be checked into a project's source control and contain enough information to reproduce the context on another machine.

Both CLPM's CLI and the client provide commands to add new constraints to a context, update an entire context so that every project is at the latest release that satisfies all constraints, or update just a subset of the projects in the context. Additionally, the CLI provides commands that can execute other, arbitrary, commands in an environment where ASDF is configured to find only the projects installed in the desired context.

Global contexts are largely inspired by Python virtual environments created by the virtualenvwrapper project [13]. Project-local contexts are largely inspired by Ruby's Bundler project [4].

## 6 USE

In this section we provide some examples of how to use CLPM. We focus on project-local contexts (bundles) as we believe these are a more broadly useful feature than global contexts.

### 6.1 Installing CLPM

Tarballs (and MSI installers) of the most recent CLPM release, along with up to date documentation, can be found at https://www.clpm.dev. Windows users merely need to run the installer. Linux and MacOS users need to only unpack the tarball in an appropriate location (typically `/usr/local/`). If you wish to install from source, the CLPM source code (as well as its issue tracker) can be found on the Common Lisp Foundation Gitlab instance at https://gitlab.common-lisp.net/clpm/clpm.

After installing CLPM, it is recommended that you configure ASDF to find the CLPM client. To do this, simply run `clpm client source-registry.d` and follow the instructions printed to the screen.

Last, you may wish to consider loading the CLPM client every time you start your Lisp implementation. You can determine the currently recommended way of doing so by running `clpm client rc` and reading the printed instructions. The remainder of this section assumes you have CLPM installed and have a REPL where the CLPM client is loaded.

### 6.2 Configuration

CLPM reads its configuration files from the `~/.config/clpm/` folder on non-Windows OSes and `%LOCALAPPDATA%\clpm\config\` on Windows. The file `clpm.conf` contains most of the configuration, but the defaults should be sufficient for this demo.

The file `sources.conf` contains a list of sources for CLPM to use when finding projects and their metadata. For this demo, it will be easiest if we populate this file with a pointer to the primary Quicklisp distribution. Place *only one* of the following forms in `sources.conf`.

This form uses the primary Quicklisp distribution directly. Note that we are fetching it over HTTPS. All of the primary Quicklisp distribution's files are served over both HTTPS and HTTP, even though the Quicklisp client itself can only use HTTP.

```
(:source "quicklisp"
 :url "https://beta.quicklisp.org/dist/quicklisp.txt"
 :type :quicklisp)
```

This form uses a mirror of the primary Quicklisp distribution. This mirror exposes the same data, formatted using the CLPI specification. Using this source will result in CLPM performing some operations faster (because it can download only the needed data). However, this source may take some time to be updated after a new version of the Quicklisp distribution is released.

```
("quicklisp"
 :url
 "https://quicklisp.common-lisp-project-index.org/"
 :type :ql-clpi)
```

### 6.3 Downloading the Demo System

We have a simple project designed for use in CLPM demos. It is located at https://gitlab.common-lisp.net/clpm/clpm-demo. Clone it anywhere on your file system and checkout the `els-21` branch.

### 6.4 Creating a Bundle

We will now create a bundle for the demo system. The simplest possible bundle is one that uses a single project index and specifies that all constraints come from .asd files. To create such a bundle you need to perform two actions. First, a `clpmfile` must be created. This can be done at the REPL by evaluating:

```
(clpm-client:bundle-init
 #p"/path/to/clpm-demo/clpmfile"
 :asds '("clpm-demo.asd" "clpm-demo-test.asd"))
```

After evaluating this, you will have a `clpmfile` that looks similar to the following:

```
(:api-version "0.3")

(:source "quicklisp"
 :url "https://beta.quicklisp.org/dist/quicklisp.txt"
 :type :quicklisp)

(:asd "clpm-demo.asd")
(:asd "clpm-demo-test.asd")
```

Notice that the first statement in the `clpmfile` declares the bundle API in use. This allows for the file format to evolve over time while maintaining backward compatibility. Second, notice that every directive is simply a plist. This makes it trivial for any Common Lisp project to read and manipulate the file.

After the `clpmfile` is created, the bundle dependencies must be resolved and installed. To do this, evaluate the following at the REPL:

```
(clpm-client:install
 :context #p"/path/to/clpm-demo/clpmfile")
```

CLPM does not believe in modifying a context without explicit permission from the developer. As such, the client will by default produce a condition before it performs any modifications. This condition describes the actions that are about to be performed and has two restarts established for it: one to approve and perform the changes, and one to abort. Therefore, upon evaluating the above form, you will be dropped into the debugger to approve the changes. Of course, this behavior can be customized.

This will create a file `clpmfile.lock`. This file should not be modified by hand. It contains all the information necessary for CLPM to reproduce the bundle on another machine, including a complete dependency graph.

### 6.5 Activating the Bundle

Once a bundle has been installed, the next step is to activate it. The activation process configures ASDF so that it can locate all systems in the bundle. The activation process also optionally integrates the CLPM client with ASDF. This allows CLPM to notice when you try to find a system that is not present in the bundle and can provide options on adding it to the bundle. This integration is enabled by default and discussed more next. You can activate the bundle by evaluating:

```
(clpm-client:activate-context
 #p"/path/to/clpm-demo/clpmfile")
```

The CLPM CLI has an equivalent command that configures ASDF using only environment variables. This feature is particularly useful when running scripts or when running tests via CLI. For example, to run an SBCL process where ASDF is configured to use only the systems in the bundle *without* also needing to have the client loaded, simply run `clpm bundle exec -- sbcl` in the same directory as the `clpmfile`.

You can now load the demo system by evaluating:

```
(asdf:load-system "clpm-demo")
```

### 6.6 Modifying the Bundle

The most common modification made to a bundle is to add more dependencies to it. If your system acquires a new dependency you have two options on how to add it to the bundle. The first option is to explicitly reinstall the bundle using `install`. This will find any new dependency and install it, while trying its best to not change the installed versions of the projects already in the bundle.

The second option is to just use `asdf:load-system` to reload your system. If you have the client's ASDF integration enabled, the client will notice that the system is missing from the bundle and take action. The default action is to signal a condition informing the developer of the situation with several restarts in place. Following CLPM's guiding principles, however, this behavior can be modified to, for example, automatically install the dependency à la the Quicklisp client.

To see this in action, open the file `clpm-demo-test.asd` and add `"fiveam"` to the `:depends-on` list. Then try to run the system's test suite by evaluating `(asdf:test-system "clpm-demo")`. Fiveam was originally not part of the bundle so ASDF signals a condition when it tries to load it. However, we can use the `reresolve-requirements-and-reload-config` restart to install fiveam and then see the tests pass.

### 6.7 Using Development Versions

Now, let's say that fiveam's author has added a really cool new feature that you want to start using before it's released into Quicklisp. You can easily do this by telling CLPM to install fiveam directly from git. Simply add the following directive to your `clpmfile`.

```
(:github "fiveam"
 :path "lispci/fiveam"
 :branch "master")
```

Then evaluate `(clpm-client:install)`. Note that you do not have to specify the context if you have already activated it. If you evaluate `(asdf:test-system "clpm-demo")` again, you can see that it loads fiveam from a different path than it did previously.

### 6.8 Developing a Dependency

Last, let's say that you would like to add a new feature to fiveam and use it in clpm-demo. You should not modify any files installed by CLPM, as they may be reused between projects and the `.git` folder is stripped from any dependencies installed directly from their git repo.

Instead, you should clone the project you want to work on yourself and tell CLPM to use that checkout instead of the one it installs. To demonstrate this, clone fiveam (https://github.com/lispci/fiveam) so that it is next to the clpm-demo repository. Then, create the file `.clpm/bundle.conf` inside the clpm-demo repository with the following contents:

```
(version "0.2")

((:bundle :local "fiveam")
 "../fiveam/")
```

This tells CLPM to not install fiveam (as you have already done it) and to use the git repository located at `../fiveam/` (relative to clpm-demo's `clpmfile`) when resolving dependencies.

A `clpm-client:hack` function that performs these steps automatically is currently being developed. It is slated for inclusion in CLPM v0.5.

## 7 CONCLUSION AND FUTURE WORK

We have presented CLPM — the Common Lisp Project Manager. CLPM introduces many features to the Common Lisp community that are taken for granted in other programming language specific package managers. Key among these features are HTTPS support, a standalone CLI to the worker, global and project-local context management, and lock files. Additionally, CLPM adds these features without forcing a particular development practice or environment and without sacrificing "Lispy-ness" or interactive development.

We plan to continue developing CLPM and continue adding useful features. Two planned features of note are an extensible architecture for client/worker communication and the ability to add scripts from an installed project to a user's `PATH`. The former would enable a myriad of new configurations, including CLPM workers deployed as persistent daemons that communicate with clients over network sockets or setups that are Dockerized or otherwise isolated from other processes on the system.

In addition to improvements to CLPM itself, we aim to continue contributing back to the upstreams of our dependencies. Notably, we are in the process of interacting with the ASDF developers to add support for more expressive version numbers and dependency version constraints in `defsystem` forms.

Last, it is our strong preference to enable developers to use the complete power of CLPM without introducing a split in the community with regards to project indices. Therefore, we would like to take ideas and lessons learned from CLPI and integrate them into Quicklisp distributions.

## REFERENCES

[1] Armed Bear Common Lisp. https://abcl.org/.
[2] ASDF – Another System Definition Facility. https://common-lisp.net/project/asdf/.
[3] Steel Bank Common Lisp. http://www.sbcl.org/.
[4] André Arko and Engine Yard. Bundler: The best way to manage a Ruby application's gems. https://bundler.io/.
[5] Alexander Artemenko. Ultralisp - Fast Common Lisp Repository. https://ultralisp.org/.
[6] Firejail Authors. Firejail Security Sandbox. https://firejail.wordpress.com/.
[7] Zach Beane. Quicklisp. https://www.quicklisp.org/beta/.
[8] Eitaro Fukamachi. Dexador. https://github.com/fukamachi/dexador/, .
[9] Eitaro Fukamachi. Qlot. https://github.com/fukamachi/qlot/, .
[10] Python Software Foundation. The Python Package Index. https://pypi.org/.
[11] Chad Fowler, Rich Kilmer, Jim Weirich, et al. Rubygems. https://github.com/rubygems/rubygems.
[12] Nicolas Hafner. Common Lisp Portability Library Status. https://portability.cl/.
[13] Doug Hellmann. virtualenvwrapper. https://virtualenvwrapper.readthedocs.io/en/latest/.
[14] SANO Masatoshi. Quicklisp-HTTPS. https://github.com/snmsts/quicklisp-https/, .
[15] SANO Masatoshi. Roswell. https://github.com/roswell/roswell/, .
[16] Yukihiro Matsumoto. Ruby. https://www.ruby-lang.org/.
[17] The pip developers. pip - The Python Package Installer. https://pip.pypa.io/en/stable/.
[18] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697.