ChuMP and the Zen of Package Management

Nicholas Shaheed
nshaheed@ccrma.stanford.edu
CCRMA, Stanford University
Stanford, California, USA

Ge Wang
nshaheed@ccrma.stanford.edu
CCRMA, Stanford University
Stanford, California, USA

> I can confirm that I'm not unreasonably excited about package management.
>
> —Author 1

> Can we even write a paper about a package manager?
>
> —Author 2

Abstract

ChuMP stands for "ChucK Manager of Packages", designed to automate the process of installing, upgrading, and removing software components for the ChucK programming ecosystem. ChuMP manages libraries, tools, audio and graphics plugins in a centralized, structured, and versioned manner. This project originated out of the recent ChucK development "renaissance" alongside a growing user community, now entering its third decade. The time for package management, as the ChucK slogan goes, is now.

What began as a practical project has expanded into broader reflections on tool-building, service, and community. As we labored on what seemed like a "no-brainer" tool that everyone wanted but that no one wanted to build, questions arose: "how did we get here?", "what is the role of service-based tool-building in our field–and what, if any, is its research value?"—in short, "can we even write a paper about a package manager?" Meanwhile, we couldn't help but notice that the act of creating a package manager seems to unify not only disparate software fragments, but also something of community. In other words, there may be more than meets the eye. This paper chronicles the making of a package manager and all that goes along with it. This is the story of ChuMP.

Keywords

Package manager, philosophy, craft, community, thankless work, "is this research?"

1 Introduction

> **Package manager**: a collection of software tools that automates the process of installing, upgrading, configuring, and removing computer programs for a computer in a consistent manner.
>
> —from Wikipedia, the free encyclopedia

1.1 What is ChuMP?

ChuMP is a package manager for the ChucK music programming language. It is a tool that queries and installs and upgrades plugins and libraries from a centralized package repository for ChucK and its growing ecosystem of community-supported tools.

NIME '25, June 24–27, 2025, Canberra, Australia

Since its inception more than two decades ago, ChucK—a strongly-timed computer music language[16]—has been used for sound design, instrument building, audiovisual design, games, laptop orchestras, live coding, composition, performance, as well as many educational contexts. Somewhat worryingly, ChucK has even found application in healthcare[13].

While ChucK is in use within and beyond different academic institutions, the amount of cross-pollination has tended to be minimal: tools created by users historically have remained decentralized and confined to their respective silos. As the ChucK ecosystem and community have experienced renewed growth in recent years[14], the need for the discovery, sharing, and distribution of commonly-used tools has increased drastically. A package manager seemed to be the logical answer to address this need.

While everyone seemed to want a package manager, nobody seemed particularly excited to build one. Through an unlikely sequence of events, the authors *willed* a package manager into existence. Moreover, what began as a project of pure practicality has inadvertently ballooned into a meditation on tool-building, service, community, and even the nature of academic research. The sheer mind-numbing dryness of designing and implementing a package manager has compelled the authors to ponder its very philosophy. Is this even research? (Or can it at least *appear* to be research?!) The ecosystems of peer computer music languages already have package managers,[1][2][3] and yet there are no research papers in the computer music literature regarding these tools. What is the academic value of making something that serves an obvious need, but may not be readily publishable?[4] Can a package manager create new contexts and enhance community? Should we have, like, not written this paper?

In order to unpack these and others questions, we will need to start at the beginning.

2 Origins of a ChuMP

(Section 2 is written as a personal account of Author 1.)

2.1 Original Sin; or, Temptations of Package Management

Back in spring of '22, after taking two or three computer music composition courses at Stanford University that were taught in ChucK, I began to get involved in ChucK development at

[1] https://docs.cycling74.com/userguide/package_manager/
[2] https://github.com/pure-data/deken
[3] https://doc.sccode.org/Guides/UsingQuarks.html
[4] a thought experiment: is academic publication a form of package management?

CCRMA. At the time, it was a small coalition of two or three people making infrequent updates to the core language. I came into the project with software engineering experience, a healthy interest in programming languages, and as a composer who has worked enough with the language to compile a mental list of features I wanted as part of my own compositional work and that I thought could make the language better in general.

"Why is it so hard to reuse code?", I found myself wondering on many occasions. Beyond the limited examples included in the distribution, it was difficult to find existing code, even as I had the vague sense that ChucK is being used in many places. The code was out there, but also out of reach. When I did happen to find code that looked useful, a familiar pattern emerged: I would copy and paste it into my giant ChucK file that represented whatever piece I was working on. Hundreds and sometimes thousands of lines were scattered around a chaotic soup of text where it would become increasingly impossible to find anything. This was not helped by the fact that miniAudicle–ChucK's official IDE, a 15-year old piece of software–did not have basic search functionality on Windows. This was symptomatic of a broader disease, as many areas of ChucK have needed attention since the mid-2010s. At the time, ChucK lacked everything from class constructors to a compile-time import system to even up-to-date API documentation. Multiple reasons contributed to the stagnation of ChucK's development, including Author 2—the chief architect of the language–devoting his efforts elsewhere during that time: as a tenure-track professor in a mobile music startup company who also ended up writing a comic book on designing tools such as ChucK. There was some irony in the latter.

Meanwhile, I was jealously eyeing the ecosystems of Max/MSP, and not-specifically-audio environments such as Processing and Python. Each of these had their own package manager, and accompanying websites and documentation, that drastically increased the discoverability and access to vast libraries of plugins and libraries. Yet, I was not ready to give up on ChucK, a text-based computer music language that also offered unique (and sometimes wacky) language features such as strongly-timed concurrency. And so I kept on ChucKin'.

The precursor project to ChuMP was not a package manager, but an effort by Author 1 and Author 2 in 2022 to add a compile-time import system into the language. This functionality would allow a ChucK program to statically import the contents of another, a feature that was shockingly absent in the language. Having this would enable ChucK code to be shared modularly and from which a package manager would benefit. Unfortunately, this project quickly failed as we realized the undertaking would entail sweeping updates to the compiler and type system. The ChucK core codebase at the time was quite the hot mess. We were not ready.

But soon, everything changed in ChucK development.

2.2 The ChucK Renaissance

The formation of the ChucK Kitchen Cabinet (CKC) took place organically over the next year. This group of ChucK developers at CCRMA (and remotely) form the core of ChucK development. Independent efforts to improve ChucK coalesced into regular meetings, sprints, and development "hackathons". Today ChucK development is an active research initiative with over a dozen members at CCRMA alone. In less than two years, the CKC has drastically "resurrected" ChucK, ushering in a new era of expansion, experimentation, and tool-building in what has been dubbed the "ChucK Renaissance."[14]

During this period of renewal, the ChucK ecosystem grew: WebChucK[9] and WebChucK IDE[3], ChuGL (ChucK graphics)[17], ChAI (interactive AI)[5], Chunity (ChucK in Unity)[1], SMucK (Symbolic Music in ChucK)[4], and more. Significant additions to the core language were introduced: class constructors and an import system (finally), an overhaul of the compiler, type, and runtime systems, proper API documentations[5]—all of which drastically expanded the ability to create reusable code libraries and plugins. ChucK development was accelerating at a rate not seen in over a decade.[6]

An *intangible* outcome of the CKC was a sense of shared purpose and community: "...the CKC is a group of individuals each with their skills and interests to bring to bear on various aspects of the language. In this setting, there is both structure and freedom for all the members, providing a balance of function and fun. This is perhaps the closest ChucK development has come to a formal and centralized development process and is certainly the largest by group size." [14] Meanwhile, the renewed and sustained development has spurred a growth of the ChucK user community.[7] Truly, "if you build it, they will come."

With all of these advancements and expansions, it seemed that the stars had aligned—for a package manager.

2.3 A Thing That Everyone Wanted but No One Wanted To Build

On the last day of the inaugural weekend-long ChucK-a-Thon, during our future planning meeting, the topic of package management resurfaced. There was plenty of enthusiasm for having a package manager, and as a group we came up with a rough list of features we wanted to see. But, after all this discussion and enthusiasm, the big questions was dropped: Who was going to make the thing? An awkward silence overtook the room. As we looked at each other expectantly, the unspoken sentiment was clear, a package manager for ChucK was a thing that everyone wanted but no one wanted to build. It sure sounded like a time-consuming project that would take away from our composing, our classes, and research we were actually excited to do. But still, it would be such a great thing to have. For reasons I can no longer recall, I tepidly spoke up and agreed to do it, as if saying, "I'll take the ring to Mordor!" People were quick to latch onto my "initiative" with enthusiasm, as the realization dawned that I had just committed to making a package manager.

2.4 Who is the Real Chump?

(Answer: Author 1)

As established, I did not *really* want to make this package manager, same as everyone else. But I pressed forward and tried to plan out what exactly this package manager was going to be. I was suddenly faced with a barrage of design questions: What even was a package? How do I design this so that it can work with command-line interface (CLI) ChucK, miniAudicle, WebChucK, etc? What language features are needed to get this working? What about dependencies? Versions?! How am I going to coordinate developers to submit packages? Due to decision paralysis and my general lack of enthusiasm for the project, I began procrastinating. This part was easy. There are always

[5] https://chuck.stanford.edu/doc/reference/
[6] despite this, to date miniAudicle still does not support search on Windows
[7] join the ChucK community Discord server! https://discord.gg/Np5Z7ReesD

Figure 1: The ChuMP logo. Made in Microsoft Paint, intended to mimic drawing with a crayon.

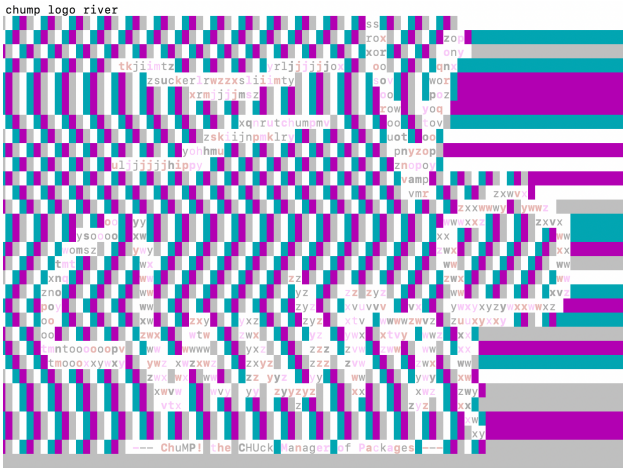

Figure 2: `chump logo river` - a frame from one of four animations based on the ChuMP logo.

other deadlines, and plenty of work that was more interesting, and more fun to do. So progress was slow. The time that I did dedicate to working on this project was mostly spent fretting over an endless series of *what-ifs*. There was a lot of thinking and planning, but very little doing. I was stuck, in desperate need of something, anything that would get me coding.

2.5 What is in a Name?

As 2023 was nearing a close, Author 1 and Author 2 met to discuss requirements for the package manager. Eventually, we turned to the topic of what to name this theoretical package manager. Almost instinctively, we were leaning towards a questionable pun, as is the style of most ChucK projects (Chunity, ChAI, chugins, ChuGL, etc.). Mostly, we were trying to move around "Ch", "P" (for Package), and "M" (for Manager) in various combinations. Soon enough, the name "ChuMP" was proposed, and then immediately we applied a hamfisted backronym to it: the ChucK Manager of Packages. There was something I found immensely charming about this silly, mildly crass name: the cludgy acronym, the question of "who is the chump here?" (see Section 2.4). All of it was wonderfully ridiculous when attached to something as seemingly dull as a package manager. The coining of ChuMP filled me with motivation I had not felt since committing to the project. Quickly afterwards, we created a Github repo titled simply "chump,"[8] a logo was made (Figure 1) and development finally began in earnest.

2.6 Crouching Tiger, Hidden Motivations

This burst of motivation coming from something ridiculous and/or unrelated to the immediate task at hand became a trend in ChuMP development. As part of an effort to improve the appearance of ChuMP (dubbed Operation: ChuMP Vanity), we experimented with ANSI codes to colorize—among other things—the download progress bar and the help page. With these practical visual improvements, things were becoming easier to navigate; it also looked nice. Like a kid suddenly given a bag of fireworks with no adult supervision, my mind began to wonder, "What can I do with *all these colors*?"

[8] https://github.com/ccrma/chump

I spent the rest of the day and part of the next day putting together a series of terminal animations based off of an ASCII-fied version of the ChuMP logo (Figure 2). Nobody asked for these. Of course I immediately stuck them in the main branch, and added the subcommand `chump logo` with a menu of animation styles (`breakfast|cereal|river|dim`). As I admired my handiwork, I was confronted by a truth that I had known but did not want to admit to myself: creating a package manager is tedious work; any joy from this thankless task will depend on my ability to distract myself from the required work, in new and increasingly elaborate ways.

Over time, the sum of these "useless" "distractions" did make development easier. There was the terrible idea of embedding a full ChucK virtual machine inside ChuMP to sonify package management; or ways to map a package download to a ever-descending Shepard-Risset Tone. We didn't go through with all of them, but they filled me with imagination and a strange sense of motivation. After two months of sustained development, I could see the light at the end of the tunnel. And one day, I looked over what I had done and found myself saying, "This is ChuMP."

3 The Way of ChuMP (A User Manual)

ChuMP is a CLI tool available for Windows, Mac, and Linux. It supports a set of obligatory actions often associated with package managers: install, uninstall, update, list, and info, which are invoked by calling `chump <subcommand>`.

`chump install <Package>`, unsurprisingly, downloads and installs a package. ChuMP will automatically identify the latest version of the package that is compatible with your operating system and ChucK version. Many brain cells died in order to bring this feature into the world. If you want to install a specific version of a package call `chump install <Package>=<VersionNo>`.

`chump uninstall <Package>` does the opposite. It uninstalls a package.

`chump update <Package>` will update an installed package to the latest version.

`chump list` will list all available packages.

`chump list -i` will list only currently installed packages.

`chump info <Package>` will list detailed information about a specific package.

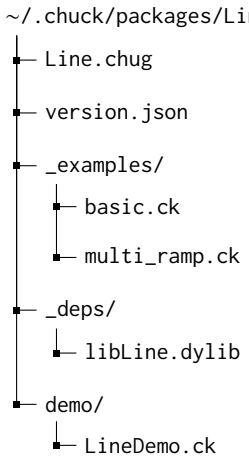

Figure 3: An example of the directory structure of a package

`chump logo <Mode>` will play a logo animation. Try these, they are fun and they gave Author 1's life meaning during ChuMP development.

`chump help` for more info on ChuMP.

Say that you want to install the `Line` package, which includes the `Line` UGen. First you would call `chump install Line`. Now you can import `Line` from any ChucK file on your system:

```
// import Line.chug from the Line package folder
@import "Line"

// Line is now in the type system and can be instantiated
SinOsc foo(880) => Line bar(5::second) => dac;
```

4 What is a Package?

A ChuMP package can contain any combination of ChucK code, chugins, and data files (e.g., audio, text, images, models). Once installed, a ChuMP package is located as a folder inside a designated package directory. On MacOS and Linux, this is `~/.chuck/packages`, on Windows it is `C:/Users/%USERNAME%/Documents/ChucK/packages`. This folder houses `version.json`, containing metadata about the package, and any number of files and directories.

There are several reserved package subfolders: `_data/`, `_examples/`, `_docs/`, and `_deps/`. These are mostly used for documentation generation purposes or are being reserved for future use. However, `_deps/` is specifically for including dynamic library dependencies (i.e. `.dylib` or `.dll` files) for chugins (a ChucK-plugin, a compiled binary written in C++). If a chugin is loaded from a package's directory, then ChucK will load any dynamic libraries found in the `_deps/` folder. This addresses a long-standing issue of managing libraries in a portable and distributable way being quite cumbersome. Particularly in Windows, where any `.dll` dependencies not explicitly loaded in the executable have to be placed in the same directory as `chuck.exe`, which requires admin permission to do. What used to be a tedious process of diagnosing runtime errors and manage system dependencies is now handled with a simple `chump install` and `@import` in ChucK code.

4.1 Defining Packages

ChuMP was designed such that packages can be created and submitted by anyone in the user community, subject to testing and approval by a ChuMP authority (probably Author 1). In order to create a new package to be managed by ChuMP, one can use `chumpinate`, a developer-facing library that lets you define and generate packages inside of ChucK (that can alos be installed using ChuMP). This will generate the necessary metadata that resides inside a package. For details on how this works, see Subsection 5.1.

5 Technical Stuff No One Cares About

5.1 A Deep Dive into Defining Packages

All ChuMP package metadata is stored in a `manifest.json` file that is fetched from a centralized server. Package listings are managed in a GitHub repository,[9] and the `manifest.json` is automatically generated from this repository. Each package consists of a JSON file defining metadata about the package—name, authors, description, etc. Every package has one or more *releases*. A release is a specific package version with a specific set of files that are downloaded and installed when `chump install` is called. If a release does not include any compiled binaries, such as one that only has Chuck files, then a new package version will only have a single release associated with it. If a package includes compiled code that is OS and CPU-architecture specific (i.e. a chugin compiled for x86-64 Windows), then a package version will have multiple releases associate with it—one for each OS/architecture combination. For MacOS, universal binaries are supported so potentially only one release is needed for MacOS, rather than two.

A declarative API is provided for defining packages and releases in ChuMP. It is available under the package Chumpinate (`chump install Chumpinate`). This lets package developers define packages, and create a release. The script will then output three artifacts: a `package.json`, defining the top-level package metadata, a `release.json`, defining the specific release, and a `Package.zip`, a zip file containing all the files of the release that is structure in a way that ChuMP can handle automatically when installing. The two JSON files are then added to the package repository, and the ZIP file is uploaded to a server to be accessible at the URL specified in the Chumpinate script. An example script can be found in Appendix A.

5.2 Versioning

ChuMP package *releases* must have a defined version, which follows the `Major.Minor.Patch` semantic versioning scheme (i.e. `v1.2.3`). When a user attempts to install a package, ChuMP uses this system to automatically determine the newest package that is compatible with the user's system and installed ChucK version.

5.3 ChuMP Implementation

ChuMP was mostly implemented in C++17 and assembled using the Meson build system. It consists of the core ChuMP library (which handles the core logic of package installation, updating, etc.), a CLI wrapper around this core, the Chumpinate chugin which adds the capability of writing ChuMP package definitions inside of ChucK, and several helper scripts, including one to injest the package repository, and generate a new manifest file. There

[9] https://github.com/ccrma/chump-packages

are also provisions for automatically generating a web-based front-end for the database of packages.

ChuMP relies on several libraries, including libcurl (for downloading files), minizip-ng (for zipping and unzipping packages), nlohmann_json (to read JSON files), and OpenSSL (to perform file hashing). It uses Catch2 for testing, and there are a suite of continuous integration tests built using GitHub Actions.

(The terseness of this subsection is not proportional to the amount of boredom and consternation it has inflicted.)

5.4 The Journey of a Package

When a user calls `chump install Pkg`, the `manifest.json` file is read, and parsed into a structured list of Packages, each of which contain a list of releases. ChuMP then tries to find `Pkg` in the Package list. If it does, it then tries to find the highest-versioned release that is compatible with the user's operating system, CPU architecture, and ChucK version. If this succeeds, then ChuMP will proceed to go through the list of files in that release, download them, validate their checksums, and then move them into the proper directory.

When a user calls `chump uninstall Pkg`, ChuMP validates that the package is currently installed. If it is, it will delete all files associated with the package. If the `Pkg/` directory is empty after this, it will be deleted as well. If it is not (i.e. there are files in the directory not associated with the package), ChuMP will leave those files there.

`chump update Pkg` is similar to `chump install`. But, before an update is applied, all files associated with the old package release will be removed from the user's system.

5.5 Design Considerations

Max/MSP and Pure Data both have GUI-based package managers. SuperCollider has the Quark system, which takes advantage of the language's dynamic typing and object-oriented design: packages are objects that are installed and managed in-language (with an optional GUI). ChuMP's interface instead takes inspiration from more general-purpose package managers such as apt-get, DNF, and pip, which are command line programs. This is both a practical decision—CLIs are easier to implement, generally more portable, and are more easily scriptable than their GUI counterparts—a reflection of the preferences of Author 1, and a strategic architectural choice: from the outset ChuMP was designed with the various ChucK IDEs in mind (miniAudicle and the WebChucK IDE). A core ChuMP libraray wraps all the internal logic and data structures, and an interface is built as a wrapper around that library. The CLI tool was the first wrapper, but future plans include implementing a GUI package manager (likely similar to Max/MSP's) inside miniAudicle and the WebChucK IDE.

Two distinct advantages of GUI-based package managers compared to CLI tools is ease of use and discoverability—no knowledge of how to use a terminal is needed and, i.e. with Max/MSP's package manager, browsing and discovering new packages is straightforward and installation is a single button click. In order to help bridge this gap, the package listing website[10] takes inspiration from the Homebrew online package browser[11] and provides both an accessible way to browse available packages and gives users the proper installation command which can simply be copy and pasted into the terminal.

[10] Available ChuMP Packages: https://chuck.stanford.edu/release/chump/
[11] https://formulae.brew.sh/

5.6 Storage and Distribution

As of ChucK version 1.5.5.0, ChuMP is now packaged as a standard tool in the main language distribution for macOS and Windows (Linux users will need to build from source). Currently, all packages and related files are hosted at, and distributed from, CCRMA's web servers, along with the rest of the ChucK website. Initial packages include a mixture of C++ chugins (e.g., FluidSynth and Rave) and native ChucK libraries (e.g., SMucK, a new symbolic music framework for ChucK). A full list of available packages at the time of publication is given below:

- **FluidSynth**: A UGen for loading and rendering soundfonts
- **SMucK**: A framework for symbolic music notation and playback in ChucK
- **Rave**: A UGen to load and synthesize real-time audio from variational autoencoder models. Based on IRCAM's RAVE (Real-time Audio Variational autoEncoder) by Caillon and Esling[2].
- **PlinkyRev**: A stereo reverb UGen ported from the Plinky synth[12]
- **Rec**: Helper functions for recording to audio files. Supports recording from dac, UGens, and arrays of UGens.
- **WarpBuf**: A UGen for high-quality time-stretching and pitch-shifting of audio files; also supports Ableton .asd files
- **Hydra**: A wrapper for the Python configuration framework Hydra
- **Line**: A UGen for creating envelopes of arbitrary ramps (ala Max/PD's line object)
- **Patch**: A tool for updating control values from UGens
- **Chumpinate**: Two classes (Package and PackageVersion) to faciliate creating new packages for ChuMP

6 Reflections

> The true package manager will package Man.
>
> —Absolutely no one

6.1 Community

Through the process of making ChuMP, what began as a purely technical project that was addressing an obvious need, had morphed into a social one as well. In hindsight this was rather inevitable, having committed to this project in a communal space—the CKC hackathon. But, as Author 1 emerged from the solitude of development, with a partially functioning package manager, another issue arose. A package manager needs packages! In a quest to make this software useful in practice, Author 1 began reaching out, first to others in the ChucK Team, and eventually to people in the broader ChucK community. By necessity, Author 1 connected with people and their work that spread across the far-flung corners of the ChucK world. From creating a universal binary in order to more easily distribute the notoriously finicky FluidSynth chugin that had been in regular use by the Stanford Laptop Orchestra (SLOrk), to bundling dozens of Faust effects compiled down to chugins, to projects from the broader ChucK community, such as the impressively comprehensive LiCK library by Michael Heuer.[13]

These social interactions were refreshing, mostly. It was good to talk to people now that ChuMP actually exists and functions—and to work together towards common aims, even if some of the short-term aims include deciphering mysterious CMake dependencies in order to get a package to build, or updating hundreds of

[12] https://plinkysynth.com/
[13] https://github.com/heuermh/lick

older ChucK files for a single package in order to take advantage of the latest language features.

At the time of writing, another ChucK Hackathon had recently taken place. It was during this weekend, filled with XXtra Flammin' Hot Cheetos and Chinese takeout, that these communal interactions permeated this event with excitement and anticipation. No one quite talked about it, but it had been during a ChucK Hackathon some two years earlier that Author 1 begrudgingly agreed to making a package manager. ChuMP had come full circle.

6.2 *Who Cares?* On The Question of Audience

Upon reflection, it seems the target audience of ChuMP is really the ChucK community, which can be further distilled into three roles. Candidly, the initial impetus for ChuMP came from ourselves, the centralized ChucK development team at CCRMA, and our need to practically manage a rapidly growing set of tools and components within the ChucK software ecosystem. Secondly and more obviously, ChuMP intends to benefit potentially any ChucK user by allowing them to incrementally incorporating new tools into their work. Thirdly, ChuMP was created as a tool for third-party developers within the community to distribute their work. For a computer music language ecosystem with a strong emphasis on open-source software, ChuMP serves as both *glue* and *hub*.

6.3 Hard-hitting ChuMP Testimonials

Firstly, we have a user testimonial from Celeste Betancur, composer, performer, long-time ChucK user. When asked *how did that make you feel, chump-wise?*, she replied "Hahahaha chumpifyed in the most satisfactory possible way." (She had recently used ChuMP to prepare for a live coding performance.)

Another testimonial comes from Kelly Cochran, a member of the teaching team of the Stanford Laptop Orchestra (which undergoes software updates regularly). "...FluidSynth installed on all the [SLOrk laptops] :) chump is amazing." This sentiment is perhaps warranted, given that in the past, finding, building, and installing the finicky FluidSynth chugin was no small feat. Here was the typical process: we would realize we'd want to use FluidSynth for a new piece. We track down the source code, fighting with temperamental build systems, updated compiler versions, and new operating systems idiosyncrasies—only to give up hours later. After a resurgence of optimism and caffeine, we would manage to produce a macOS build, but only when linked against non-portable Homebrew libraries. Installing the FluidSynth chugin entailed creating a shell-script that copied both the chugin as well as all runtime dynamic library dependencies onto each machine. While this brittle setup worked in practice, no one was quite confident this would not break if anything were changed on these machines, and we knew in our hearts that we would need to repeat the process in the future. Now, all of that has been simplified to just typing `chump install FluidSynth`.

Sometimes, the availability of a tool can directly change, in practice, what people create and learn. RAVE (a controllable real-time variational autoencoder for audio synthesis) was used by the Stanford Laptop Orchestra in its exploratory first assignment, including some on-the-fly `chump installs` of RAVE when moving between laptops. This would not have happened without ChuMP (the students would have simply used another available tool, which would have been fine, too).

6.4 Service and the Zen of Toolbuilding

Service may be defined as an intentional act that results in direct benefits more for the community than for the individual doing the service. In other words, building a thing that everyone wanted but that no one wanted to build. Author 1 has noted that the time spent developing ChuMP is likely to far outweigh the cumulative time-saved in using ChuMP over his lifetime. Author 2 tends to feel similarly about ChucK as a whole. And yet, we did it, and mostly do not regret it. Why *is* that?

Perhaps it has to do with *craft*, which the ancients more or less defined as a practice (e.g., flute-playing) that 1) you care to get better at for intrinsic reasons ("I want to know I can do it!") and 2) can get better at doing only through practice (e.g., reading 1000 books on flute playing is no substitute for actually picking up the flute and playing it). Furthermore, there is a joy in the making and perhaps even in the maintenance of tools. These notions reflect the ethos with which ChucK—yet another computer music programming language that nobody quite asked for—has been developed for the past 20 years. And while we genuinely endeavor for ChucK to be useful, there has always been the hidden, if sometimes deferred, satisfaction of working on a tool that we know inside-out, down to its nuts and bolts.

ChuMP, in a sense, may be a pure product of this ethos. It is a tool that we knew to be a thankless job—in other words, a service. It is also a tool that we (correctly) assumed would be tedious in its implementation. And in undertaking this task anyway, we have extended a tradition of craft, one that speaks to some inner need to build tools not only to have tools, but to improve our capacities for such endeavors. To borrow a sentiment from *Artful Design*, "what we make, makes us." [15] Despite the often boring and probably-unpublishable nature of a package manager, we intrinsically cared about the *quality* of the tool. We wanted to make ChuMP "good", starting with considering what "good" even means for a package manager.

A chump has worked thanklessly, and now we have a working ChuMP. Strangely, there is a feeling of satisfaction (it comes and goes). The ChucK ecosystem has been improved by ChuMP, with the possibility that before long, ChuMP could well become an everyday part of any ChucK user's workflow, however briefly. There is also a satisfaction in knowing that we won't have to ever write a packager manager for ChucK again. Hopefully.

If we take a step back, there resides the possibility, hidden among all the unglamorous details, for a virtuous cycle in which community and tool-building reinforce one another. ChuMP was born out of the ChucK Team—and now it has created new contexts and interactions within the group. As for the larger ChucK user community, the hope is ChuMP becomes a tool that people use and that also *stays out of their way* and allow them to do the work they want to do (this would imply ChuMP is doing its job—a "good" ChuMP).

The name of this paper is adapted from Robert Persig's book *Zen and the Art of Motorcycle Maintenance: An Inquiry Into Values* [11], which meditates not only on the notions of craft and quality, but on the value of *all* components that comprise a complex system; even a thing seemingly as mundane and insignificant as a single screw in a motorcycle, in the right contexts, can be critical to the functioning of the whole system. Maybe a package manager is a kind of screw, or something that manages screws.

6.5 Academic Paper Writing

What is the academic value of building tools that serve an obvious practical need, but that are of questionable novelty? And is it novelty we are truly after in our work? At the same time, it seems good to at least recognize that tools like package managers support (and can even influence) our research and the art we make. How do we compass *that*?

In the constellation of computer music fields, we do encounter papers that mention package managers. For example, "Disperf: A Platform for Telematic Music Concert Production", discusses the potential benefits of a package manager as part of its Future Work [10]. At the same time, and as far we can tell, there are no papers specifically on computer music package managers; the closest seems to be publications on large-scale software distributions such as PlanetCCRMA[6] and CARL[7].

Somewhat surprisingly, our literature review has unearthed academic publications on package managers outside our "immediate orbit," some of which argue for the role of package managers. The authors of LuaRocks write, "While sometimes dismissed as an operating systems issue, or even a matter of systems administration, module management is deeply linked to programming language design." and "language-specific package managers have risen as a solution to these problems, as they can perform module management portably and in a manner suited to the overall design of the language[8]." In this view, creating language-specific package managers that are cognizant of the "ways of thinking and doing" associated with a specific language amounts to more than a standalone tool, but also becomes a part of the language, its ecosystem, and community.

So, in writing what may be the very first paper on a package manager in our field, are we advocating for a world brimming with academic publications on package management?

Maybe?

Or, at least, *why not?* What if such papers are capable of addressing aesthetic, craft-based, and/or communal implications of package managers (or similarly thankless but useful tools)? This line of questioning extends an argument made in Ramsay and Rockwell's widely read article, "Developing Things: Notes Toward an Epistemology of Building in the Digital Humanities[12]", a rumination on the scholarly value of building tools. One of its takeaways is that scholarship of a work should transcend its format (e.g., peer-reviewed journal articles) and be evaluated also on the authors' willingness and effectiveness to critically consider their work (whatever the format, including software tools) with respect to broader contexts in rich and provocative and entangled ways. And if such connections can be drawn between a package manager and its implications for process, craft, and community, should we not—for all intents and purposes—consider building such tools to be research?

https://chuck.stanford.edu/chump/

7 Acknowledgments

The authors would like to thank/blame the ChucK Team for causing this work to happen, in particular Mike Mulshine, Kelly Cochran, Celeste Betancur, Alex Han, and Kiran Bhat. Thanks to Walker Smith for encouraging us to write (too) honestly. We thank the ChucK User Community and the Stanford Laptop Orchestra for being early adopters of ChuMP. We are thankful and relieved that the reviewers saw something in this paper, even when we might not have. We would also like to acknowledge that throughout this paper, the word "chump" appears 137 times, including this appearance, and accounts for 2.3% of its words. Thank *you* for reading this far.

8 Ethical Standards

ChuMP has been developed with the support of CCRMA's departmental funding, curricular student research, and volunteer contributions. The authors are aware of no potential conflicts of interest.

References

[1] Jack Atherton and Ge Wang. 2018. Chunity: Integrated Audiovisual Programming in Unity.. In *NIME*. 102–107.
[2] Antoine Caillon and Philippe Esling. 2022. Streamable neural audio synthesis with non-causal convolutions. *arXiv preprint arXiv:2204.07064* (2022).
[3] Terry Feng, Celeste Betancur, Michael R Mulshine, Chris Chafe, and Ge Wang. 2023. WebChucK IDE: A Web-Based Programming Sandbox for ChucK. *proceedins of Sound and Music Computing* (2023).
[4] Alex Han, Kiran Bhat, and Ge Wang. 2025. SMucK: Symbolic Music in ChucK. In *Proceedings of the International Conference on New Interfaces for Musical Expression.*
[5] Yikai Li and Ge Wang. 2024. ChAI: Interactive AI Tools in ChucK. In *New Interfaces for Musical Expression.*
[6] Fernando Lopez-Lezcano. 2002. The Planet CCRMA software collection. In *ICMC.*
[7] Gareth Loy. 2002. The carl system: Premises, history, and fate. *Computer Music Journal* 26, 4 (2002), 52–60.
[8] Hisham Muhammad, Fabio Mascarenhas, and Roberto Ierusalimschy. 2013. LuaRocks-a declarative and extensible package management system for Lua. In *Programming Languages: 17th Brazilian Symposium, SBLP 2013, Brasília, Brazil, October 3-4, 2013. Proceedings 17.* Springer, 16–30.
[9] Michael R Mulshine, Ge Wang, Jack Atherton, Chris Chafe, Terry Feng, and Celeste Betancur. 2023. Webchuck: Computer music programming on the web. In *New Interfaces for Musical Expression.*
[10] Michael Palumbo, Doug Van Nort, and Rory Hoy. 2020. Disperf: A Platform for Telematic Music Concert Production. In *Proceedings of the 2020 International Computer Music Conference. Santiago, Chile.*
[11] Robert M. Persig. 1974. *Zen and the Art of Motocycle Maintenance: An Inquiry Into Values.* William Morrow and Company.
[12] Stephen Ramsay and Geoffrey Rockwell. 2012. Developing things: Notes toward an epistemology of building in the digital humanities. *Debates in the digital humanities* (2012), 75–84.
[13] Babak Razavi, Pooya Ehsani, Maryam Kamrava, Leonid Pekelis, Vaibhav Nangia, Chris Chafe, and Josef Parvizi. 2015. The brain stethoscope: a device that turns brain activity into sound. *Epilepsy & Behavior* 46 (2015), 53–54.
[14] Marise van Zyl and Ge Wang. 2024. What's up ChucK? Development Update 2024. In *Proceedings of the International Conference on New Interfaces for Musical Expression.* 549–552.
[15] Ge Wang. 2018. *Artful Design: Technology in Search of the Sublime.* Stanford University Press.
[16] Ge Wang, Perry R Cook, and Spencer Salazar. 2015. Chuck: A strongly timed computer music language. *Computer Music Journal* 39, 4 (2015), 10–29.
[17] Andrew Zhu and Ge Wang. 2024. ChuGL: Unified Audiovisual Programming in ChucK. In *Proceedings of the International Conference on New Interfaces for Musical Expression.* 351–358.

A Appendix: Creating a ChuMP Package

Here is an example ChucK program that constructs a MacOS release of the Chumpinate package. The release can then be generated by calling `chuck make_release.ck`

```
// Import the Chumpinate package, which adds both
// "Package" and "PackageVersion" to the type system
@import "Chumpinate"

// instantiate a Chumpinate package
Package pkg("Chumpinate");

// Add our package metadata...
"Author 1" => pkg.authors;
"https://github.com/chump/chumpinate" => pkg.repository;
"https://chump.com" => pkg.homepage;
"MIT" => pkg.license;
"Two classes (Package & PackageVersion) to help create
 packages to be used with ChuMP (the ChucK Manager
 of Packages)" => pkg.description;
["util", "chump", "packages"] => pkg.keywords;

// generate a package-definition.json
// This will be writte to "Chumpinate/package.json"
"./" => pkg.generatePackageDefinition;

// Now we need to define a specific release of chumpinate
PackageVersion rel("Chumpinate", version);

// The current version of the package
"0.1.0" => string version;

// Because this is a chugin, it is compiled against
// specific chugin API headers. The chugin API Version
// must match the API version of installed ChucK
"10.2" => rel.apiVersion;

// There must be a minimum compatible language verison.
// Optionally, you can also set languageVersionMax.
"1.5.4.0" => rel.languageVersionMin;

// Specify which version and architecture this
// release is compatible with
"mac" => rel.os;
"universal" => rel.arch;

// The PackageRelease.addFile(...) and related methods
// compile a list of files that will be zipped up in the
// final Release.zip file.

// The chugin file. This goes in the top-level
// directory of the package
rel.addFile("../Chumpinate.chug");

// These build files are examples as well. These
// go in the _examples/ directory.
rel.addExampleFile("build-pkg-win.ck");
rel.addExampleFile("build-pkg-mac.ck");
rel.addExampleFile("build-pkg-linux.ck");

// Documentation files. These go in the _docs/
// directory
rel.addDocsFile("./index.html");
rel.addDocsFile("./chumpinate.html");
rel.addDocsFile("./ckdoc.css");

// wrap up all our files into a zip file,
// and tell Chumpinate what URL
// this zip file will be located at.
rel.generateRelease("./", "Chumpinate_mac",
 "https://chump.com/Chumpinate/" + rel.version() + path);

// Generate a version definition JSON file, to
// be added to the package listing repo.
rel.generateReleaseDefinition("Chumpinate_mac", "./" );
```