The Ways Of Vellum

Automation Design Patterns For "Making The Sausage"

Zed A. Shaw

Apr 2008

Preface

This book will teach you about Vellum, and indirectly teach you about automating your daily work as a programmer. I've found over the years that automating what I do all day long is the single quickest way to improve my efficiency. Finding that tool or tiny script that makes mundane tasks go away is very rewarding. Sometimes, the automation makes you a super hero in the eyes of programmers who missed out on this secret.

When I wrote Mongrel I used Rake to automate nearly everything I did. I ran unit tests, deployed it, started servers in clusters, thrashed it with fuzzing, and even built the core of Mongrel with a parser generator named Ragel. Using tools to create code that I'd normally make by hand made it possible for me to crank out a complete web server for multiple Ruby web frameworks in a matter of months.

Yet, I still see programmers–myself included–avoiding automation as if it is a burden. They groan when they're asked to make a script to administer a system. They whine when there's a chance to turn deployment into a solution. Daily task automation is seen as a boring task similar to taxes and accounting.

I think bad tools are to blame for programmers (and system administrators especially) avoiding ruthless automation in their daily work. Take a look at Make and its followers. Ever tried to list the available targets in a Makefile? How about just the wide range of different configuration files many systems have (if they have those even). Tools like cfengine are so complicated you need a Ph.D. in Theoretical Cognitive Science to make them work reliably. Everything about the business of removing the human is blocked by gross interconnects realized as ugly configuration files. No wonder programmers avoid it.

For years I've wanted to make a build tool. Call me weird, but I've always enjoyed removing useless work and the useless people who surround easily automated useless work. Vellum is the start of that build tool. It's especially important because it first focuses on a task that has always been hairy and ugly: Automating the build of books with LaTeX. The idea is that if Vellum can make tangling together a book about software (like this one) then it can probably automate just about anything.

Building books isn't the only thing Vellum does, it's just the first problem I solved with it. In the process of automating the creation of this book, I also used Vellum to crank out builds and releases of Vellum itself, Idiopidae, my website, and many other open source projects I created. Vellum hasn't failed me yet (although, that's because I just fix it to not fail).

Hopefully you like my little book, and since it's free I will gladly refund your purchase price to you.

Typography

I used quite a few tools while making this book, including Vellum itself. Each of these tools was designed specifically to help in the creation of books "by programmers for programmers". I release all of these tools as GPLv3 licensed projects that anyone is free to use to do something similar. [1]

Here's the big list of all the things I used when preparing this book, so you can judge how well it fits in with your own mode of operation. [2]

TeX Live A very well put together and community maintained version of TeX (and LaTeX).

Idiopidae My project for mixing code and prose together. Think of it as a compiler for people who have to write about code.

Vellum Well, the build tool this book is all about.

Pygments A very nice Python library that converts code to various formats. Used to make the very nice code samples.

ArchLinux My main desktop machine. Sucks for media, but awesome for code.

Awesome Window Manager Someone who's fucking awesome should use a fucking awesome window manager. That window manager is Awesome.

Vim The entire set of software was written using Vim, including all the software projects and the LaTeX source.

evince A very good PDF and PS viewer for working on LaTeX documents

IPython The IPython shell is a great alternative to Python's default REPL, but it has this great feature where you can merge into it regular shell operations by running `ipython -p pysh`. This is weird at first, but eventually it really grows on you having Python and shell all in the same terminal merged together. In the examples below when you see a shell it's a cut-paste from an IPython shell session. This is also why you might see some Python typed into the shell where appropriate.

[1] The works you create obviously do not need to be licensed under the GPL.
[2] Yes, this combination is weird, but as you'll see later that's on purpose.

Unusual Conventions

I started Vellum shortly after PyCon2008 so I could work on a book, but not just any book, the Great American Code Novel. I wanted my book to be about the code for a project, and that meant making the code easy to include in the book and look gorgeous. While working on the problem of including code, I found that even the venerable TeX sucked for writing books with lots of code.

After months of trying various tools and reading the writings of other people who've worked on the problem I came to the realization that I should just write something. Then at PyCon at a dinner with many other Addison/Wesley authors, we came this conclusion:

> Every author writes a "book making" tool set, says it sucks, and never releases it.

With that in mind I became determined to create a set of tools to help programmer authors document their gear. The problem of building a book about code could be solved with some simple tools for building, formatting, and sharing, but it also needed more research and experimentation.

While writing this book I came to realize that a number of TeX features are actually quite irritating for books about code. For books about mathematics they're great, but for code they make following the discussion difficult.

When reading this book, keep these typographical "conventions" in mind, as they're different from the ones you find in other books:

There are no floats of code in this book. A float is a logical table or figure that relies on TeX's algorithms to determine placement. When discussing a piece of software this is annoying because the prose becomes disconnected from the section of code being discussed. Instead, in this book the code is just merged into the prose to look like a float, but it's actually placed right there.

Code is syntax highlighted . All code is formatted with color in the PDF form the way it would be in many editors. If you print it the book may look wrong or be impossible to read.

Sections of referenced code have logical names. A major problem when reading through a book with lots of code is finding a snippet later. You can find all the code in this book via a simple format of **file.ext:section** and each code block has a more descriptive comment to aid in searching. If you need to find in the Vellum source where **press.py:1** is located, then it's the first line of **press.py**. If you need to find where **press.py:load** is then search for *export "load"* in **press.py**.

Code sections are bounded by graphics. Each code section ends with a ↩ in the right margin and has line numbers in the left margin. They also have a fancy oval box headers with all the code snippet's information like the file and section and short description.

All code runs and is tested. The code is actually fully functional and taken out of an active project, so if it differs from the prose then the code is right.

Otherwise, I let LaTeX format everything the way it wanted, with just a few logical modifications to highlight *identifiers* and **files**.

Terminology

There are some simple terms you should know before reading this book, but if you're experienced at build automation then you can skip this section.

build tool A tool that automates some process in a particular order. Just about anything can be a build tool.

declarative logic This is a style of logic where, instead of telling a computer *how* to do something you tell a computer *what* to do. That's the classic definition, but many times this means translating incremental reasonable processes into obnoxious convoluted tree structures just to keep a computer happy.

lists and dicts Python has two primary data structures of lists and dicts. In other languages these might be called "arrays and hashes" or "lists and hashtables" or even just "associative arrays" for both.

Quick Tour

This book is organized from least detailed to most detailed information on using Vellum to get things done. There isn't theory in Artificial Intelligence or declarative logic programming systems. It just tells you how to get your job done in the cleanest nicest way.

Since many times you approach Vellum only when you really get tired of typing commands and you've got very little time to stop and sharpen your knives, I've

done you a favor and described all the chapters. Start wherever you want, although the book is small enough you could read it all and be a grand master.

Introduction This chapter right here, I hope you like it.

Starting If you have to read just one chapter before your whole company catches on fire, then this is the one. You can probably do 90% of your work with just this chapter.

Running More of a reference chapter that covers all the command line options, what they do, available commands, and how to do common tasks like find targets that mention a command.

Building In depth coverage of the Vellum syntax, how to structure a build, debugging the build, and making your build specifications modular. It also gives a complete example of doing a complex build.

Debugging Helpful tips and tricks for figuring out what's wrong with a Vellum build.

Extending Advanced extension techniques for creating your own commands and packaged recipes. This is the chapter to read if you want to standardize your builds for other future projects.

Embedding Quick overview of the API and implementation of Vellum then a sample of embedding Vellum's engine in your own software. In this case we make a simple distributed build tool.

Contributing Simple instructions on the Vellum code standards, a crash course in Bazaar and Launchpad, and how to contact me (Zed) to give back.

Appendix A: The Code A full in depth run through Vellum's code so you can get started hacking on it. The code is small so this is a fairly easy tour.

Appendix B: Converting From Make A case study of converting from CMake and regular Make on two projects.

If you're in a hurry and want to get automating right away then just go to Chapter 2. It has enough information, and as you work you can refer to Chapter 3 to figure out how Vellum works.

More serious build commanders will want to read the whole book from front to back. It's not a large book so it shouldn't take you long.

Contents

Chapter 1

Introduction

Putting software together is an ugly task. Down right disgusting in some parts of the world. Depending on the size of the project you could be dealing with chains of files that must be carefully coaxed through various compilers, linkers, loaders, analyzers, and generators. Any tiny misstep and the software is busted. This is just to run the program to see if your changes worked, not even to package it for sales and marketing to shove off to people.

I've been building software for years, and on every project I've created the same bags of custom shell scripts and "Make-like-files" to automate the build and deployment. Whether the task is just compiling everything or pushing out new deployments the end result is the same: A high pile of crap. What I've always wanted was a simple solution that could eliminate or greatly reduce the dung pile on my next project.

I thought that solution was Rake, until I used it on a fairly large enterprise project initially run by a bunch of morons. What I found is that Rake's easy access to Ruby meant that all the problems of Ruby meta-programming were there in my builds. At any moment I could be hit by a landmine of stupid where someone reshuffled the build without telling me. In one case, I spent the better part of 3 days figuring out that some jerk had just replaced the default Rails test task with his own special sauce that didn't work. He did it with a monkey patch[1], in a hidden directory, many levels down, and no comments.

What I wanted from a build tool is something tiny and easy to understand, no real magic in it, not entirely declarative, and with consistent build specifications between projects. This meant that upon joining a new team you could understand the build without having to troll through mountains of code deciphering some idiot's version of Prolog[2]. It also meant the tool needed to have a set of constraints that kept things consistent without causing too much inconvenience.

Vellum is my attempt at this tool. It is still a work in progress, but it does encode all the various bits of knowledge and experience I've gained from building things. Not so much the actual writing of software, but the mundane dumbness of putting all the pieces other people wrote into something a regular Joe can use. Just a simple tool that a programmer can understand in an evening and start using right away[3].

[1] That's an evil practice in the Ruby world where novices try to be like their heroes and do all software modification via indirect surgery on the class hierarchy rather than just plain old objects.

[2] If you've ever used Capistrano, you know what I mean

[3] Because we all know programmers never use anything without reading the code first.

1.1 Why Software Automation Sucks

I've been fascinated by why system automation, particularly building software, is so ugly. It seems whenever you need to automate a task that involves different systems the resulting software *must* violate every software design best practice. Whether that's a data munging process, an automated deployment, or a simple C project build.

I blame the state of system automation on four converging factors that amplify the suckitude of the whole situation:

1. It's always a last minute afterthought done out of necessity.
2. It's different every time for every person on every project.
3. The tools available are just too damn hard to understand.
4. There is no "Automation Design Patterns" book to use as a guide for best practices.

This lack of planning, repeatability, and simplicity with no experienced guide book turns what should be a fairly simple automation task into something very difficult.

Let's analyze venerable Make as the proposed "tool to end all build tools" for a minute:

1. It was written in the 70's to help build complex compiled language programming projects.
2. It uses a weird declarative execution process that is file centric and so convoluted it can't even list the available targets you can use.
3. It assumes that the result of every command execution is a single output file.
4. There are many different incompatible versions of Make that all have slightly different features.
5. There are no good books for Make other than an ancient O'Reilly book.
6. It has no standard construction so every build is totally different for every project.
7. You don't have access to a real programming language.

In all honesty I've *never* seen a nice Makefile. They're always ugly, convoluted, confusing, and hard as hell to trace through. Make doesn't help because it's missing features you'd need to analyze a Makefile like listing available targets, searching for targets containing commands, or even just visualizing the *whole* structure includes and all. Instead, every person who debugs a Makefile ends up running targets and scanning through includes looking for some messed up spacing on a dependency that causes silent errors.

Make and all the similar build tools are stuck in an age where just getting a program to run was a painful thing involving many shell commands. People were very happy to just get a tool that helped stage and organize this alone. In fact, a Makefile is almost analogous to the plain old interpreter used by modern languages like Python, Ruby, Scheme, Lisp, Lua, and Forth. Makefiles are the C programmer's REPL[4] and interpreter.

If it's the case now that modern programming languages don't necessarily need a tool like make, then what do they need?

[4] Read-Evaluate-Print Loop

1.2 What Is Vellum Really?

Vellum is my attempt at a modern tool for automating modern software projects. If you're a hard-core C programmer working on 100,000,000 lines of C code with a mix of C++ for the GUI and Perl holding it all together then Vellum is not for you. If you're starting a new project in a nice language that is interactive and doesn't require endless compile processes then Vellum might be just you need. In this latter case it means your problems are now more about project management and automating implied social process your team has (then using those processes on further projects).

Let us compare the environment for make's genesis and what we have now:

1. Modern languages have some form of REPL and self-compiling behavior for fast interactive testing and programming.
2. Many languages have extensive test suites that are run easily without build steps.
3. Modern scripting languages are easily accessible from within the build tool so no excuse for not providing access to the raw language.
4. Simply building a binary is not enough, the build tool now has to update bug lists, project pages, build manuals, automatically check out from a source repository, do automated regression suites, *and* all reliably.
5. Did I mention source repositories? When make was invented there weren't too many of these, and those that were there sucked badly or weren't even network aware.
6. It's a scary world now full evil hackers [5] looking to make a name for themselves. Trusting a giant pile of autoconf or even a full programming language like in SCons and Rake is dangerous. Not being able to easily look for bad things doesn't help.
7. Software tools are more complex producing various outputs and using multiple inputs. LaTeX alone produced close to 9 files just when making this book. This makes using a file-centric build tool like make a huge pain since it assumes that one file comes out of one file going in. Even SCons has problems with this.

With these changes in the environment software is built I believe it's time to explore new ideas for building software. Some of these ideas are:

1. Why spend so much time and effort detecting file level changes when modern languages just run what you hand them and unit test suites are all or nothing?
2. Why can't a tool let me analyze its structure to look for targets, find commands, view extensions, and be self-documenting?
3. How can a tool help sites that accept project builds create the software more safely?
4. More important, how can the tool help an end user be more safe when downloading and source-building software?

[5] I use "hacker" in the sense of a person breaking into a computer for criminal purposes. The fashionable term "cracker" is racist to white people.

5. Do build tools seriously need to be huge and confusing with half-assed attempts at Prolog? Can they just be small and easily understood?

6. Does a build tool really need every target in a massive logic tree structure? Can't I just tell it to do stuff out of order sometimes?

7. Can a tool balance the need to extend it with new features and the need for people using the resulting build to feel comfortable that some junior idiot didn't hide evil or go crazy with his "mad skillz"?

8. What's the minimum syntax needed to describe a build specification?

These questions are obviously loaded, but they're the ones that I asked while writing Vellum. It will give you an idea of what I was trying to get done with the tool.

1.3 Vellum And Python Philosophy

Vellum is written in Python and was one of the projects I started to learn idiomatic Python usage and philosophy. Python is great in that they have their philosophy of mind easily accessible:

> import this
>
> The Zen of Python, by Tim Peters
>
> Beautiful is better than ugly.
> Explicit is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> Flat is better than nested.
> Sparse is better than dense.
> Readability counts.
> Special cases aren't special enough to break the rules.
> Although practicality beats purity.
> Errors should never pass silently.
> Unless explicitly silenced.
> In the face of ambiguity, refuse the temptation to guess.
> There should be one–and preferably only one–obvious way to do it.
> Although that way may not be obvious at first unless you're Dutch.
> Now is better than never.
> Although never is often better than *right* now.
> If the implementation is hard to explain, it's a bad idea.
> If the implementation is easy to explain, it may be a good idea.
> Namespaces are one honking great idea–let's do more of those!

1.4 Features At A Glance

This is handy because I can give Vellum a similar list that matches up with the ones Python people *claim* to follow:[6]

Beautiful is better than ugly. The build specification format is a very simple and easy to type syntax that is consistent but it is *not* a full programming language. Don't worry you can easily escape into Python when you need by writing new command modules of your own or new recipes.

Explicit is better than implicit. Everything about Vellum is as explicit as possible without being obnoxious. This means that instead of the weird "file-centric" mode found in other build tools where files are targets, Vellum makes you explicitly specify the files involved. Targets are instead actually named things people use to make Vellum do stuff.

Simple is better than complex. Vellum is less than 1000 lines of code with all the docstring comments, and about half that if you remove those. That's including the parser, command modules, process execution, and all command line options.

Complex is better than complicated. I have no idea what the hell they mean here. That's the dumbest thing I've ever heard.

Flat is better than nested. Vellum encourages a flatter structure within the confines of a simple namespace system, and then it collapses the names you use into a set of small dicts structures so you can easily find things.

Sparse is better than dense. Velum has special syntax for running commands and most build specifications are fairly small with very little symbols in them.

Readability counts. I designed Vellum's format for readability such that you can scan down a file and quickly spot what's going on. I also took the dependency chains and moved them to their own section so that there's only one place you look in each file to see how targets are structured. This also makes build files more modular since targets are not dependent on the build structure to operate.

Special cases aren't special enough to break the rules. Vellum tries to unify everything into lists, dicts, strings, and commands which are consistent Python functions in a consistent location.

Although practicality beats purity. But, it still is Python so you can just do whatever the hell you want in your commands, and probably even use other build libraries if you want.

Errors should never pass silently. Vellum has very good error and debugging features only ignoring errors if you tell it to (while still printing them out).

Unless explicitly silenced. Yes, you can also tell Vellum to shut-up with the `--quiet` option.

In the face of ambiguity, refuse the temptation to guess. Vellum's internal processing algorithm is so simple that it can't be smart enough to guess what you want.

[6] While I admire the fact that Pythonistas have this creed, and for the most part it helps keep them all on the same frequency, there's all sorts of places they arbitrarily violate this list of ideas.

There should be one–and preferably only one–obvious way to do it. I rarely find the *obvious* ways to do thing in Python all that obvious, but Python does have very good documentation which helps you figure out what "obvious" means. Most of the time however, Vellum's structure is very simple once you learn a few rules, and everything else can be inferred from that.

Although that way may not be obvious at first unless you're Dutch. I'm not Dutch so I wrote this manual for everyone to learn about Vellum.

Now is better than never. Uh, Vellum builds stuff now rather than never.

Although never is often better than *right* now. I kind of like my build tools to build things *right* now so Vellum does that.

If the implementation is hard to explain, it's a bad idea. This book has a whole appendix that leads you through the code so you can learn it, but the core processing is easily understood in a few paragraphs of text.

If the implementation is easy to explain, it may be a good idea. Well it was easy to explain so I'm guessing it is a good idea.

Namespaces are one honking great idea... Finally, Vellum has the concept of imports where it brings in commands called "modules" and other build specifications called "recipes". When it does this is puts them in a namespace so that you can reuse them in other builds without having to modify much.

Following the Python philosophy while building Vellum was a good learning experience. It taught me about Pythonic software design, but also about how the Python community reacts to ideas they think don't follow the philosophy given above.

1.5 Vellum's Not Make

Make has one huge advantage over Vellum: *history*. There's so much institutional knowledge about building software using various linkers and compiles that Vellum would never be able to compete. Add in autoconf and you've got probably 20 years of black art hidden deep in a mess of M4 macros that I don't even want to go near.

While it wouldn't be difficult to add configuration commands and commands to automate building most C or C++ projects, I'd say just go use make. Use the simplest make you can to get the thing built, then use Vellum to automate all the other drudgery and crap you have to deal with after that.

Chapter 2

Starting

Every programmer needs a place to start that's real and visceral. If you actually waded through all the talk so far then either I'm a very good writer or you're very obedient. I prefer to jump to the chapter that shows me the money.

This is the chapter that shows you the money. It will get you up and running with the minimal amount of theory and get your first build going in no time. You'll have to learn a tiny bit of how Vellum structures its world, but after that you are good to go.

I suggest that you try to make a build using this chapter first, and try playing with the results and breaking things. After you have everything working and are comfortable using it then move on to the more advanced later chapters.

2.1 Installing Vellum

Thanks to *Easy Install* you can install Vellum with one command:

Code: easy install vellum – *Using Easy Install*

```
zedshaw@monstrosity[~]|119> sudo easy_install zapps pygments idiopidae vellum
Searching for zapps
...
Installed /usr/lib/python2.5/site-packages/zapps-0.X-py2.5.egg
...
Installed /usr/lib/python2.5/site-packages/Pygments-0.X-py2.5.egg
...
Installed /usr/lib/python2.5/site-packages/idiopidae-0.X-py2.5.egg
...
Installed /usr/lib/python2.5/site-packages/vellum-0.X-py2.5.egg
Processing dependencies for vellum
Finished processing dependencies for vellum

```

After that you should have it installed and it should be ready to use. Doesn't get much simpler really.[1]

If you don't have Easy Install or you don't have access rights to install it, or you just plain want to do it the old fashioned way, then you can install it using **setup.py**. Since the purpose of this chapter is to get you up and running with Vellum quickly I won't go into this option. If you want to use this option then you probably know what you're doing already.

2.1.1 Minimalist Install

You don't have to install Idiopidae or Pygments if you just want to run Vellum. Those two programs are used by the Vellum build to make the book, so I'm having you install them now for later use where you'll be running the Vellum build to explore the design.

If all you want to do is install Vellum then you just need Zapps:

Code: minimal install vellum – *A Minimal Install*

```
zedshaw@monstrosity[~]|119> sudo easy_install zapps vellum
Searching for zapps
...
Installed /usr/lib/python2.5/site-packages/zapps-0.X-py2.5.egg
...
Installed /usr/lib/python2.5/site-packages/vellum-0.X-py2.5.egg
Processing dependencies for vellum
Finished processing dependencies for vellum

```

In reality you could strip the installation down even further since the only thing Vellum needs from the Zapps project is the **zapps/rt.py** file.

2.1.2 Testing The Install

You can then test the install by simply running **zapps** without options, and **vellum -h** to see their help. Hopefully if everything is installed right then the rest of this tutorial will go smoothly.

2.2 Your First Build

We're going to start off small and do a little toy build so you can get a good understanding of the structure of a Vellum build specification (from now on just referred to as a "build spec"). We'll start by having it do a hello world example and then expand it to automate all sorts of other tasks, print out commands, explore the build, and then at the end we'll do a practical example with a real Python project.

Vellum works by default on a file named **build.vel** in the current directory. It loads this file, processes any imports and arguments you give, and then it runs the targets you specify. There's some extra things Vellum can do since the build spec is a simple internal data structure and not actual code, but this is the majority of its operation.

[1] At least until you need to uninstall it.

To get started, make a directory somewhere and then put this into a file named **build.vel** in that directory:

Code: build.vel – *Baby's First Build*

```
options(default "hello")

imports[]

depends()

targets(
    hello
        $ echo 'hi'
)

```

This simple file demonstrates the Vellum build format, which does not have many symbols on purpose. The entire file is treated as a dict, with each of *options*, *imports*, *depends*, and *targets* treated as keys and the things after them are data. Right away you should be able to see that there are "dict like sections" and "list like sections" that contain strings and what looks like shell commands.

Take a closer look again and then run *vellum* to see it say 'hi'.

Code: vellum output:1 – *Vellum Default Target Output*

```
zedshaw@timmy[chapter2]|2> vellum
BUILDING: ['hello']
-->: hello
 sh: "echo 'hi'"
hi

```

You can see that vellum ran *sh: echo 'hi'* but how did it know to do this? It's because you specified in *options* that the *default* was *"hello"*.

2.2.1 Vellum's Execution Algorithm

As a programmer you can get by without knowing how Vellum processes your file, but a simple understanding will make debugging a broken build easier. Vellum's main algorithm is actually pretty simple:

1. Load the build.vel and import any imports.
2. Merge everything together in one master dict, maintaining namespaces.
3. Look for a default target specified in *options* or use those listed on the command line.
4. Process all *depends* to build the ordered list of targets for each given target, producing a master list and remove and sequential duplicates.

5. Start at the beginning of the target list, find each target in *targets* and run it after processing templates.

6. Skip any targets that don't exist, exit when you're done.

That's literally the entire algorithm, minus any loading of externally defined commands, looking for errors. In fact, you can read the code that builds the dependency list in one function:

Code: script.py:resolve depends – *How Vellum Resolves Dependencies*

```
28      def resolve_depends(self, root):
29          """
30          Recursively resolves the dependencies for the root
31          target given and return a list with those followed
32          by the root.
33          """
34          building = []
35          if root in self.depends:
36              for dep in self.depends[root]:
37                  if dep in self.depends and not dep in building:
38                      building.extend(self.resolve_depends(dep))
39                  else:
40                      building.append(dep)
41          building.append(root)
42          return building
43
```

The *script.py:resolve_depends* function is the main meat of Vellum since it figures out what tasks to run based on the dict you put in *depends*. There's of course other processing to get to this point and some after, but understanding this one algorithm can help you debug problems.

You can see the results of this algorithm if you add some dependencies to the sample file and then run *vellum -T*:

Code: build.vel:depends – *Now With Dependencies*

```
12  options(default "hello")
13
14  imports[]
15
16  depends(
17      hello ['py.hello']
18  )
19
20  targets(
21      hello
22          $ echo 'hi'
23
24      py.hello py "print 'hello from python'"
25  )
26
```

We added a dependency that *'hello'* needs *'py.hello'* and then there's a new *py.hello* that prints out a hello message by running the python code *print 'hello from python'*. This demonstrates a few more features where you can run python code as a command.

Now, run *vellum -T* to see how this creates a dependency chain and modifies your results:

Code: vellum -T after depencencies – *Seeing What Needs What*

```
zedshaw@timmy[chapter2]|3> vellum -T
OPTIONS:
{...}

TARGETS:
hello:        ['py.hello', 'hello']
py.hello:        ['py.hello']

DEFAULT: hello

```

Great, now ignoring the OPTIONS for a minute, you see that *hello* depends on what you expect, and that *py.hello* is just running itself. Remember that vellum ignores targets that don't exist, considering this not a failure. Instead, it's more robust to skip them and then let you see that it didn't run them.[2] This means that in the dependency list a target *MUST* have itself listed otherwise it's not working right, and it should be last.

Let's run *vellum* again and see what we get:

Code: vellum after dependencies – *Watching Dependencies Work*

```
zedshaw@timmy[chapter2]|4> vellum
BUILDING: ['py.hello', 'hello']
-->: py.hello
 py: "print 'hello from python'"
hello from python
-->: hello
 sh: "echo 'hi'"
hi

```

Now Vellum tells you it's going to run *py.hello* and *hello* in that order, it runs each and prints the results.

Before covering how the *imports* and *options* work in-depth we'll have to cover the crash course for the specification format. Right now though you should be able to build simple commands without much more knowledge of Vellum other than how to make python or shell commands and dependencies.

The only thing you're missing is how to specify a target that runs multiple commands in order, which is kind of important when you need to run multiple commands. I know, it sounds crazy, but sometimes, you just want to run a series of commands.

In vellum this is easy, just do a list with a series of *line strings* for each command:

[2] There will be an option to change this behavior later.

Code: build.vel:targets – *Multi-Line Targets & Fancy Python*

```
options(default "hello")

imports[]

depends(
    hello ['py.hello']
)

targets(
    hello [
        $ echo 'hi'
        $ echo 'hello again'
        $ date
    ]

    py.hello [
        py [
            |print 'hello from python'
            |print globals()
        ]
        $ echo "i'm a shell command"
    ]
)

```

Vellum assumes that a target with a list means to run each element of the list in order, and it assumes that anything that's a *string* or *line string*[3] is just a shell command. Notice in target *py.hello* we start a multi-line Python statement as well and run it using the | character, then right after that runs a shell command, so you can mix and match whatever you need.

Run *vellum* and *vellum -T* to see how this all runs and works. Hopefully you're getting a grasp of how dependencies interact with the targets and how to structure a target. If you wanted to make more dependencies, you'd just add another line with the name of the target and then a list of strings indicating what it depends on.

2.2.2 Available Commands

Vellum is a self-documenting system which means it can introspect on the commands it knows about, the build file, and all included modules and recipes. I want you to use Vellum to jump onto a new build on a new project, and with a few simple commands find all the dead bodies and understand the project structure. Without such introspection Vellum isn't much better than the existing make tools that can't even tell you the available targets.

To list the all of the available commands you run *vellum -C* which prints out a lot, so if you just want to know what one command does, give that command:

Code: vellum -C forall – *Help on 'forall'*

[3] Really, strings and line strings are the same internally.

```
zedshaw@timmy[chapter2]|1> vellum -C forall
forall:
    Iterates the commands in a do block over all the files
    matching a given regex recursively.  You can put anything
    you'd put in a normal target in the do block to be executed,
    and when it is executed the 'var' variable is set to the
    full path of each file.  This will also be in each task
    you transition to with the 'needs' command.

    forall assumes that you want the variable to be 'file'
    so for most tasks you can leave that option out.  If you
    nest forall expressions then you'll need to give each one
    a different name (just like in a real language).

    Usage: forall(files "*.py" var "file" do [ ... ])


```

2.2.3 Build Specification In-Depth

For a very complete description of Vellum's exact grammar refer to Section A.2 which breaks the entire Vellum **build.vel** file format down step-by-step so you can understand everything.

Right now though you just need to know enough to structure some simple builds and expand out. The rule with Vellum's build format is, "If you can do Python you can do Vellum." The syntax may seem different, but it actually started out as Python's data structure format and then I removed everything and made function calls consistent with dict syntax. Otherwise everything is the same as it is for you in Python land: dicts, lists, strings, and numbers in a file that works like a module.

For vellum, a "module" is a "recipe" and is either the root build.vel file or any .vel you import. These .vel files act as dicts (not lists as in Python) and are composed of the following "stanzas" or "sections":

options You set options that control the build and have data here. It's just a dict of other things.

imports A list of import statements that include other recipes or python modules with commands.

depends A simple dict that describes what targets depend on what targets. These are placed here rather than on the targets so that each target can also be used independently and so you can see the structure of a build in one place.

targets The actual things you can tell this build spec to do as a dict of strings, lists, or line strings.

There's a lot of concepts there, but the gist of the whole format is that you have a syntax for making lists, dicts, strings, and something called a *Reference*. The whole **build.vel** is just a big dict, so the order of the above elements doesn't matter. Put them wherever it makes sense for that build. **Just make sure all of them are there.**

Everything in the Vellum format is available in Python, with the exception of *line strings* but those are easy.

Here's all the data structures you can use:

reference Vellum uses the concept of a reference for every structure. It's like saying all functions can only take one parameter, but that parameter can be lists, dicts, hashes, strings, etc. A reference is started by just a *name* like you'd start a function in python with *def myfunc*. *Example:* mything(test 'those' and 'that')

dicts Almost like Python's dicts, but you use the () (parenthesis) characters to start one. This is done so that references can be like function calls and accept named parameters. The trick is that when you put references inside a dict they become key=value pairs. *Example:* (ref1 "value" key1 "value")

lists Just like in Python, you use [] you don't need commas (,).
Example: ['hi' 'there' 'joe' ['nested' 'this']].

strings You have the usual string literals with "this" and 'that', but successive strings are treated as different strings not ones that are concatenated. *Example:* "this be zed's computer" 'this is "not" his computer'

numbers You have the usual numbers, but they're loosely defined so if it looks like a number and Python can convert it then it will become a number. *Example:* 123 0.345

line strings A feature of Vellum is that the default command for a string is the *sh* command, which runs shell commands. To make writing these shell commands easier and look more like, well, shell statements Vellum supports *line strings*. They work like Python's comments in that when a '$','|', or '>' character is encountered, everything from that point to the end of the line is considered a string. You typically use '$' and '>' in shell commands since that's what many shell prompts look like, and you use '|' in longer *py* command strings so that you can see where the indent starts. *Example:* $ I go to the end as a string.

We then need to link all of these data elements together into structures you can understand. Honestly the best way for you to understand this is to write a simple **build.vel** file, slowly enhance it, and as you do run the *vellum -D* command to see how the structure maps to Python.

2.2.4 Vellum to Python Converter

With all that understood, let's use Vellum to see how this file we wrote becomes Python structures. Here's the first **build.vel** file again:

Code: build.vel:targets – *First Build Again*

```
27 options(default "hello")
28
29 imports[]
30
31 depends(
32     hello ['py.hello']
33 )
34
35 targets(
36     hello [
```

```
        $ echo 'hi'
        $ echo 'hello again'
        $ date
    ]

    py.hello [
        py [
            |print 'hello from python'
            |print globals()
        ]
        $ echo "i'm a shell command"
    ]
)

```

Now, if you run *vellum -D* you'll get a Python dump of the internal structure of the **build.vel** so you can compare the two: [4]

Code: vellum -D – *First Build As Python*

```
{'commands': {...},
 'depends': {'hello': ['py.hello']},
 'imports': [],
 'options': {'default': 'hello'},
 'targets': {
     'hello': [
         " echo 'hi'\n",
         " echo 'hello again'\n", ' date\n'
     ],
     'py.hello': [
         Reference('py',[
             "print 'hello from python'\n",
             'print globals()\n'
         ]),
         ' echo "i\'m a shell command"\n'
     ]
 }
}

```

Ignore the "commands" element of the dict and instead look at the elements after that. Notice how it matches the what you put into the **build.vel** file almost exactly. Using the rules you can see how each section was translated into a Python structure:

commands You can ignore this for now, but what you should be seeing is just a list of functions that get called like "py".

[4] You're kind of seeing under Vellum's Kimono with this command, since you can see all the loaded "commands" that it has found, but the *-D* option is designed to help you understand the structure of the file and to debug a build. Think of it as a Vellum to Python translator so your brain can get over to the dark side.

depends Notice how your syntax of `depends(hello ['py.hello'])` was converted to Python of `'depends': {'hello': ['py.hello']}`. As mentioned a dict in Vellum starts with (then you create key=value pairs with targets, and then end the dict with).

imports This is an empty list, but we'll cover more advanced importing later.

options Just like with *depends* we simply set a dict with all the option strings we wanted, in this case we tell Vellum the default target should be 'hello'.

targets This is where the real action happens. We can see there's two keys, 'hello' and 'py.hello'. The 'hello' target just seems to be a simple string, and from out description above this will be run as a shell command of *echo 'hi'"*. The 'py.hello' target is even weirder though, because it's this *Reference* class. We'll cover that next, but that's how Vellum knows to run the command *py* with the given string of Python to run.

Reference is a class defined in **vellum/parser.py** that does the job of holding a *NAME expression* pair for later analysis. Inside a dict the *reference* grammar production actually is reused to do key=value pairing. On its own the *Reference* becomes a piece of structure used much like a function call.

In the above, you can see where you used the *py* command and passed it the Python to run. You should also be able to see how that gets translated into Python code strings inside the *Reference* object that is printed out. Your use of | characters translated into a list of strings with the exact text you wrote, no escapes necessary, which helps when formatting what Vellum should be doing.

2.3 Expanding The Build

2.4 Dealing With Distribution

2.5 Generating Documentation

2.6 Limitations

2.6.1 GNU GPL v3

Chapter 3

Running

Chapter 4

Building

4.1 Specifications In Depth

4.1.1 Symbols

4.1.2 Grammar

4.1.3 Syntax

4.1.4 Structure

4.1.5 Examples

4.1.6 Warnings

4.2 Processing Model

4.2.1 Other Build Tools

4.2.2 Vellum's Model

4.2.3 The Python Influence

4.3 Commands

4.3.1 Builtin Commands

4.3.2 Writing Your Own

4.3.3 Shell Commands

4.4 Recipes

4.4.1 Local Recipes

4.4.2 Global Recipes

4.5 Templates

4.6 Putting It All Together

4.6.1 Structing The Spec

4.6.2 Setting Options

4.6.3 Organizing Depends

Chapter 5

Debugging

Chapter 6

Extending

6.1 Writing Recipes

6.1.1 Recipe Safety

6.2 Writing Modules

6.2.1 Module Safety

6.3 Distribution Techniques

6.4 Finding Other's Toys

Chapter 7

Embedding

Chapter 8

Contributing

Appendix A

The Code

Vellum is simple enough for you to enjoy reading the code in an hour. To help you with this task I have organized the code into digestible chunks for you to read through casually while sitting near a fire. You can learn everything about Vellum in less than a few hours and *never* be mystified about its operation again.

The overall structure of Vellum follows a chain of interacting objects that do a particular job and are similar to a compiler, but with much funnier obscure names to keep things interesting.

→ **bin.py** **bin.py** is a module that encapsulates all the logic for the command line options and is where the world starts going through ...

↪ **Parser** to read the **build.vel** file to create the initial data structure composed of *lists*, *dicts*, *strings*, and *Reference* objects which ..

↪ **Press** cleans up and performs all imports for ...

↪ **Script** to pull structure out, resolve dependencies, and perform semantic checking for ...

↪ **Scribe** to run the resulting final *Script* according to dependencies and *options* settings as well as running any ...

← **commands** functions which take either a raw expression or keyword arguments depending on how it is used in the **build.vel** files.

A.1 bin.py

I hate this file, but it's what you need to write in order to process command line options from the user in Python. I'll just quickly go through the highlights and then leave it to you to read the real code.

Code: bin.py:options table – *Command Line Options Table*

```
### These options are wired up by parse_sys_argv.
options = [
 ("-f", "--file", "filename",
  "Build file to read the build recipe from (no .vel)", "store", "build"),
 ("-q", "--quiet",  "verbose",
  "Tell Vellum to shut up.",  "store_false",  True),
 ("-d", "--dry-run",  "dry_run",
  "Dry run, printing what would happen",  "store_true",  False),
 ("-k", "--keep-going",  "keep_going",
  "Don't stop, build no matter what",  "store_true",  False),
 ("-T", "--targets",  "show_targets",
  "Display the search of targets and what they depend on",  "store_true",  False),
 ("-F", "--force",  "force",
  "Force all given conditions true so everything runs",  "store_true",  False),
 ("-D", "--dump",  "dump",
  "Dump the build out to a fully coagulated build.",  "store_true",  False),
 ("-s", "--shell",  "shell",
  "Run the vellum shell prompt.",  "store_true",  False),
 ("-I", "--install",  "install",
  "Create the ~/.vellum directories.",  "store_true",  False),
 ("-v", "--version",  "show_version",
  "Print the version/build number.",  "store_true",  False),
 ("-C", "--commands", "list_commands",
  "List all commands and their help, or one.", "store_true", False),
 ("-S", "--search", "search_commands",
  "Search commands with a regex.", "store_true", False),
 ("-w", "--watch", "watch_file",
  "Watch a file and run the targets whenever it changes.", "store", None),
]

```

The options table is a succinct way to encode all the options that Vellum has without resorting to tons of repetitive code. Vellum always uses the same option setup with a pairing of short (-w) and long (`--watch`) formats and these are always available in the final *options* dict that builds can access.

This table is processed by **bin.py:parse_sys_argv** using the *OptionParser* class available in Python.

Code: bin.py:parsing the options table – *Less Code For Options Parsing*

```
def parse_sys_argv(argv):
    """
    Expects the sys.argv[1:] to parse and then returns
    an options hash combined with the args.
    """
    parser = OptionParser()
    for opt, long, dest, help, action, default in options:
        parser.add_option(opt, long,
                dest=dest, help=help,
                action=action, default=default)
    cmd_opts, args = parser.parse_args(argv)
```

```
58      return cmd_opts.__dict__, args
59
```

These options are used by the primary **bin.py:run** function to determine which of the other methods to run.

Code: bin.py:the start of the world – *Read This First*

```
175 def run(argv=sys.argv):
176     """
177     Main entry for the entire program, it parses the command
178     line arguments and then runs the other methods in this
179     file to make Vellum actually work.
180     """
181     options, args = parse_sys_argv(argv)
182
183     try:
184         script = Script(options["filename"], options)
185         opts = script.options
186         if opts["show_targets"]: show_targets(options, script)
187         elif opts["dump"]: dump(options)
188         elif opts["install"]: install(options, script)
189         elif opts["shell"]: shell(options, script)
190         elif opts["show_version"]: print VERSION
191         elif opts["list_commands"]: commands(options, script, args)
192         elif opts["search_commands"]: search(options, script, args)
193         elif opts["watch_file"]: watch(options, script, args)
194         else: build(options, script, args)
195     except vellum.DieError, err:
196         print "ERROR: %s" % err
197         print "Exiting (use -k to keep going)"
198         sys.exit(1)
199     except vellum.ImportError, err:
200         print "%s\nFix your script." % err
201
```

The rest of the file is simply one method for each command line option and any supporting methods they need. Most of the methods are small, but if you want to add new options, this is where you go.

In the *run* function you can immediately see that a *Script* object is loaded, and if you look at the other functions you'll see they either use this *Script* object to perform analysis or they load a *Scribe* object to do some actual processing.

The most complex example of using *Script* to analyze the build is *search*:

Code: bin.py:implementation of -S – *How Searching Works*

```
112 def search(options, script, regex):
113     """
114     Searches through all available targets for anything that matches the
115     given regex(es) in their name or their commands.
```

```
116     """
117     search = re.compile("^.*(" + " ".join(regex) + ").*$")
118 
119     commands = [cmd for cmd in script.commands
120                 if search.match(repr(cmd))]
121     imports = ["- %s" % imp for imp in script.imports
122                 if search.match(repr(imp))]
123     depends = ["- %s %r" % dep for dep in script.depends.items()
124                 if search.match(repr(dep))]
125     targets = ["- %s\n\t%s" % (n,pformat(b))
126                 for n,b in script.targets.items()
127                 if search.match(repr((n,b)))]
128 
129     print "SEARCH FOR: ", regex
130     if not (commands or imports or depends or targets):
131         print "\tFound nothing."
132 
133     if commands: print "COMMANDS:", commands
134     if imports:  print "\nIMPORTS:\n", "\n\n".join(imports)
135     if depends:  print "\nDEPENDS:\n", "\n\n".join(depends)
136     if targets:  print "\nTARGETS:\n", "\n\n".join(targets)
137 
```

However, the most important function which moves on to the next process is the *build* method:

Code: bin.py:building with Scribe – *Beginning of Build Processing*

```
107 def build(options, script, targets):
108     """Builds the targets."""
109     scribe = Scribe(script)
110     scribe.build(targets)
111 
```

I'm sure this file could be generalized more, but it's simple enough now and not the important part of Vellum. A major change that could happen is to use the options that are set to run a given method, rather than the large if-statement.

A.2 Parser

The *Parser* lives in **parser.py** and is probably the place where you should start for understanding Vellum's internal structure. While **bin.py** doesn't actually mention *Parser* it is used as the beginning of processing by *Press* as mentioned at the beginning of Appendix A.

The *Parser* controls how Vellum understands the user's demands, and then the *Scribe* is how Vellum follows those demands. Understanding those two pieces means you understand the core of how Vellum works.

Code: parser.py:1 – *Reference Class Used By Parser*

```
# Copyright (C) 2008 Zed A. Shaw.  Licensed under the terms of the GPLv3.

class Reference(object):
    def __init__(self,name, expr):
        self.name = name
        self.expr = expr

    def __str__(self):
        if self.expr:
            return "%s(%r)" % (self.name, self.expr)
        else:
            return "%s"

    def __repr__(self):
        return "Reference(%r,%r)" % (self.name, self.expr)

```

This fancy bit of Python starts off the main parser for the Vellum syntax, which is actually nothing more than boilerplate. Notice the inclusion of the Reference class which is the primary data structure for the AST apart from lists and dicts. From the earlier text you know that Vellum's syntax consists of these Reference objects, which are always a name followed by some expression. Now you can see how this is implemented...as a name with an expression.

In **parser.g:grammar** is the actual grammar defined using Zapps (a Recursive Descent Parser generator I also maintain).[1]

Code: parser.g:grammar – *Grammar for Vellum's Parser*

```

%%
parser Parser:
    ignore: r"[\r\n \t]+"
    token NUMBER: "[0-9]+[0-9\.]*"
    token STRING: '\'([^\\n\'\\\\]|\\\\.)*\'|"([^\\n"\\\\]|\\\\.)*"'
    token NAME: r"[a-zA-Z][a-zA-Z\-_0-9/\.]+"
    token LPAR: r'\('
    token RPAR: r'\)'
    token LSQB: r'\['
    token RSQB: r'\]'
    token ENDMARKER: '\0'
    token SH: r'[>$|]'
    token LINE: r'[^\n\r]+\n'
    token COMMENT: '#'

    rule input:
        {{ self.data = {} }}
        (
         reference {{ self.data[reference.name] = reference.expr }}
         | COMMENT LINE
```

[1] If you plan on fixing or augmenting the grammar for Vellum then you do so in the **parser.g** file *not* the **parser.py** file since the latter is generated by the former.

```
37          )* ENDMARKER
38          {{ return self.data }}
39
40      rule expr:
41              atom {{ return atom }}
42              | reference {{ return reference }}
43              | structure {{ return structure }}
44
45      rule reference: NAME expr
46          {{ return Reference(NAME, expr) }}
47
48      rule atom:
49          NUMBER {{ return atoi(NUMBER) }}
50          | STRING {{ return eval(STRING) }}
51          | SH LINE {{ return LINE }}
52
53      rule structure:
54          LSQB elements? RSQB {{ return elements or [] }}
55          | LPAR dictmaker? RPAR {{ return dictmaker or {} }}
56
57      rule elements: {{res = [] }} (expr {{res.append(expr)}})+
58          {{ return res }}
59      rule dictmaker: {{res = {} }} (reference {{res[reference.name] = reference.expr}})+
60          {{ return res }}
61  %%
62
```

The important lines for understanding the structure of a Vellum build are 32-60, and if we remove any extraneous code elements we have the following succinct grammar to study:

Code: parser.g:grammar – *Clean Grammar For Vellum*

```
1   rule input: (reference | COMMENT LINE)* ENDMARKER
2   rule expr: atom | reference | structure
3   rule reference: NAME expr
4   rule atom: NUMBER  | STRING | SH LINE
5   rule structure:
6       LSQB elements? RSQB
7       | LPAR dictmaker? RPAR
8   rule elements: (expr)+
9   rule dictmaker: (reference)+
10
```

When you read a grammar like this, you start at the "root" production (the things that are like *rule input: ...*) and then work your way through all the alternatives and subsequent productions. This encodes what's a tree of the grammar similar to those annoying sentence diagrams you did in school.

If I were to read this, I'd do it like this:

rule input: "Alright, this can take any number of *references* or *COMMENT_LINE* elements. Hmm, what's *reference* do?"

rule reference: "Reference seems to be a *NAME* and then **one** element that's an *expr*. Alright, weird, but let's see what *expr* is first before we pass judgement."

rule expr: "Ok, this looks like an expression type, and it can be either an *atom*, *reference* (again), or *structure*. Ah, so because *reference* can take an *expr* and then *expr* can also be another reference we can build a **recursive structure** with this grammar. Now, what's *atom* and *structure* contain?"

rule atom: "Sweet, that's just a terminal node, which is a fancy word for it has real data and that data is a *NUMBER*, *STRING* or *SH_LINE*. Looking up at the tokens I know what all of those are, although *SH_LINE* seems to be like *COMMENT_LINE* but it produces a string. Very handy, but then what's *structure*."

rule structure: "Ok, this seems to be how you construct lists and dictionaries in order to make a full data structure. List looks normal as like in Python and can take optional *elements*, but why is a dictionary encoded with parenthesis (*LPAR* and *RPAR*)? Let's look at elements first."

rule elements: "Yep, just takes as many *expr* types as possible and appends them to a list before returning the list. Pretty simple. Since *expr* can take all of this the structure is recursive so we can build up nearly anything we can in Python."

rule dictmaker: "Now why the hell does a dictionary in this grammar take a parenthesis rather than braces like in Python? That's just weird. Ok, it seems to not take *expr* elements but instead *reference* elements. Hmm, so from the grammar for *reference* I know that it's formed like *blah expression*. Hmm, so a dictionary then is something like *(foo "test" bar "flaw" sig 123)* and then looking at the code...aha! So, a dictionary uses the *reference* grammar to create *key=value* pairs with the *NAME expr* grammar. Oh cool, so if a *reference* can also take a dictionary, and dictionaries use parenthesis then a reference can look like a function call that takes named parameters! I can write *myfun(arg1 "value" arg2 "value" arg3 "value")* and have it mean the same as *myfunc(arg1="value", arg2="value", arg3="value")*. Nice and consistent."

I obviously don't speak in *highlighting* mode, but if you walk through those paragraphs and study the grammar it should give you a good idea of how to understand it. Once you have this in your head, try just running *python vellum/parser.py input* and typing expressions in to see how they translate to Python. This little gem of a testing and exploration tool is handled by a function in the footer of **parser.g**:

Code: parser.py:footer – *Testing Case for Parser*

```
129 if __name__ == '__main__':
130     from sys import argv, stdin
131     if len(argv) >= 2:
132         if len(argv) >= 3:
133             f = open(argv[2],'r')
134         else:
135             f = stdin
136         print parse(argv[1], f.read() + "\0")
137     else: print 'Args:  <rule> [<filename>]'
138
```

A.3 press.py

Vellum uses metaphors from the printing press business as the names for many of its classes and files. In **press.py** you'll find the *Press* class which knows how to use the parser to fully load a complete build specification. This is actually difficult because *Press* has to recursively load each ***.vel** and Python module specified in the *imports* section of the primary **build.vel**

Code: press.py:class Press – *Press Loads All The Data*

```
class Press(object):
    """
    A Press uses a Parser to read a build spec and then combine it
    with any given import statements.  This is the equiv. of the
    component in a interpreter that builds an AST with the parser.

    The next stage in the processing is for the Script to get the
    Press's results to analyze the contents.
    """

    def __init__(self, main, defaults={}):
        """
        Initializes the press according to the
        defaults (which come from vellum.bin.parse_sys_argv().
        """
        self.options = defaults
        self.module_source = os.path.expanduser("~/.vellum/modules")
        self.recipe_source =  os.path.expanduser("~/.vellum/recipes")
        self.recipes = {}
        self.modules = {}
        self.main = self.load_recipe(main + ".vel")
        self.main["commands"] = {}
        # we always need these commands
        self.load('module', 'vellum.commands')
        self.imports(self.main)

```

You can see that **press.py:class Press** does some extra lifting to get things going initially. The primary thing to review is how it loads the **vellum/command.py** Python module and then completes all the imports. Without this Vellum would require you to always load those common base commands.

There also needs to be a process for finding files such that a user doesn't have to specify a full path.

Code: press.py:resolve vel file – *Finds Vellum Files*

```
    def resolve_vel_file(self, name):
        """
        Tries to find a file ending in .vel with
        the name by first trying in the local directory
        and then in the ~/.vellum/recipes directory.
```

```
41
42          It will also add the .vel if one isn't given already.
43          """
44          if not name.endswith(".vel"):
45              name += ".vel"
46
47          names = (os.path.join(n, name) for n in ["./", self.recipe_source])
48          found = [n for n in names if os.path.exists(n)]
49          if len(found) == 1:
50              return found[0]
51          elif len(found) > 1:
52              raise LoadError("More than one file named %s: %r." % (name, found))
53          else:
54              raise LoadError("Did not find file named %s at any of: %r." % (name,
55                                                                             found))
56                                                                              ↩
```

I designed *resolve_vel_file* like this so that it would give you a good error message when it couldn't find the requested file in all the usual places.

The start of the recursive processing is done by the *imports* method.

Code: press.py:import – *Rolls Through Imports*

```
149     def imports(self, import_from):
150         """
151         Goes through the imports listed in import_from
152         and then merges them into self.main.
153         """
154         if not "imports" in import_from: return
155
156         for imp in import_from["imports"]:
157             args = imp.expr
158             args.setdefault("as", None)
159             self.load(imp.name, args["from"], args["as"])
160                                                                              ↩
```

The rest of the main work is done by the *load* method which recursively calls *load_recipe* or *load_module* depending on the type of import statement encountered.

Code: press.py:load – *Loads Recipes and Modules*

```
57      def load(self, kind, file, as_name=None):
58          """
59          Given a kind of 'recipe' or 'module' this figures out
60          how to load the spec or python module.  It will do
61          this recursively until it has loaded everything, and
62          do it without causing loops.
63          """
64          if kind == "recipe":
65              file = self.resolve_vel_file(file)
66              if file not in self.recipes:
```

```
67                  spec = self.load_recipe(file)
68                  self.join(spec, self.main, file, as_name)
69                  self.imports(spec)
70          elif kind == "module":
71              if file not in self.modules:
72                  cmds = self.load_module(file)
73                  self.merge(cmds, self.main["commands"],
74                             as_name=as_name)
75          else:
76              raise ImportError("Invalid kind of import %s, "
77                                "use only 'recipe(...)' or "
78                                "'module(...)'")
79
80          return self.main
81
```

There's nothing fancy going on there, other then after loading a recipe Vellum continues to process the imports.

Since Vellum can load either other ***.vel** files or Python modules, it needs two methods to do that right named boringly *load_recipe* and *load_module*.

Code: press.py:load recipe – *The Actual Recipe Loader*

```
82      def load_recipe(self, file):
83          """
84          Loads a recipe and then calls self.load to get any
85          imports that are defined.
86          """
87          if file in self.recipes:
88              return self.recipes[file]
89          else:
90              with open(file) as f:
91                  spec = vellum.parser.parse('input', f.read() + '\0')
92                  if not spec:
93                      raise ImportError("Parser error in file: %s" % file)
94                  else:
95                      return spec
96
```

Loading more recipes is very easy since *Press* already handles much of the work itself. This method just has to do the parsing. The confusing part is that *Press* uses *load_recipe* in the *__init__* so that it can bootstrap itself, and then calls *import* to do the rest.

Code: press.py:load module – *The Python Module Loader*

```
97      def load_module(self, name):
98          """
99          Loads a python module and extracts all of the
100         methods that are usable as Vellum commands.
101         It returns a dict with the commands.
```

```
        """
        if name in self.modules: return self.modules[name]

        sys.path.append(self.module_source)
        mod = __import__(name, globals(), locals())

        # stupid hack to work around __import__ not really importing
        components = name.split('.')
        for comp in components[1:]:
            mod = getattr(mod, comp)

        # now module is the actual module we actually requested
        commands = {}
        for k,func in mod.__dict__.items():
            if not k.startswith("_") and hasattr(func, "__call__"):
                commands[k] = func

        sys.path.pop()
        return commands
```

The Python module loading is an interesting hack which I'm sure can be improved with the *imp* module. First, it appends the **~/.vellum/modules** path to the Python load path so that the *__import__* call will find it.

Reading the code however, you'll notice it has to do this strange recursive find of the actual module requested. This is because Python doesn't load the module you asked for, but its entire parent chain going up. I found this hack to get around it when reading about *__import__*.

Finally, *load_module* just rolls through all the functions in the module that don't start with an underscore and puts them in the *commands* list.

The remaining functions simply support merging together the dicts found in other recipes and transforming the names to give them a fake '.' scoping.

Code: press.py:scope name – *Scopes A Name Properly*

```
    def scope_name(self, key, name=None, as_name=None):
        """Does the common name scopings used."""
        name = as_name if as_name else name
        return "%s.%s" % (name, key) if name else key
```

This makes fake scoping by flattening the name based on the given name, if an "as-name" was indicated, etc.

Code: press.py:merge – *Merges Two Dicts With Scope*

```
    def merge(self, source, target, named=None, as_name=None):
        """
        Takes the source and target dicts and merges
        their keys according to the way scope_name does
        it.  Source is untouched.
```

```
        """
        for key,val in source.items():
            target[self.scope_name(key,named,as_name)] = val

```

Uses *scope_name* to create a merged dict.

Code: press.py:join – *Joins Two Recipes*

```
    def join(self, source, target, named=None, as_name=None):
        """
        Takes two specs and properly joins them
        using the self.merge() function on all
        of the stanzas.
        """
        # first merge the common dict style stanzas
        for section in ["targets", "options", "depends"]:
            if section in source:
                target.setdefault(section, {})
                self.merge(source[section],
                           target[section], named, as_name)

```

Finally, *join* intelligently combines two whole recipes merging the dicts that require namespaces.

A.4 Script

Script does nothing more than take all the work *Press* did and resolve the dependencies for *Scribe*. This requires busting out the contents of *Press.main,* traversing the flattened tree in *depends,* and reducing them such that no target is repeated more than once in succession.

The works is started by the *resolve_targets* method:

Code: script.py:resolving all targets – *Where Script Starts*

```
    def resolve_targets(self, to_build=[]):
        """

        Given a list of targets to_build this will make
        a new list with all of the dependencies resolved.
        """
        if not to_build:
            if "default" not in self.options:
                raise("You forgot to specify a default target and didn't give one on the comma
            else:
                return self.resolve_depends(self.options["default"])
        else:
            building = []
            for target in to_build:
                building.extend(self.resolve_depends(target))
            return self.reduce_targets(building)

```

Which uses *resolve_depends* to actually traverse the flattened dependency chains into something that can be run in order:

Code: script.py:resolve depends – *Traverse The Depends*

```
28      def resolve_depends(self, root):
29          """
30          Recursively resolves the dependencies for the root
31          target given and return a list with those followed
32          by the root.
33          """
34          building = []
35          if root in self.depends:
36              for dep in self.depends[root]:
37                  if dep in self.depends and not dep in building:
38                      building.extend(self.resolve_depends(dep))
39                  else:
40                      building.append(dep)
41          building.append(root)
42          return building
43
```

Remember how I said that targets are reduced so that there are no sequential duplicates, but that a target can be run more than once if it's separated by different targets? I found that I frequently wanted an entire target's process to run, and then the entire process for the next process to follow it. This meant that while repeating a dependent target right after it ran was dumb, I still needed targets to run if they were separated by different targets:

Code: script.py:reducing targets – *Removing Duplicates*

```
59      def reduce_targets(self, building):
60          """Given a list of targets this removes consecutive dupes."""
61          last = building[0]
62          reduced = [last]
63          for target in building[1:]:
64              if target != last:
65                  reduced.append(target)
66                  last = target
67          return reduced
68
```

This code does that, and it could just be a simple *set* instead, but when a set was used targets would run less reliably by missing targets that actually should be run again. It sounds weird, but this actually makes the builds more robust at the cost of running some targets again.

Let's use an example, say I have the two targets:

build ['generate' 'compile' 'link']

docs ['generate' 'idiopidae' 'latex']

release `['docs' 'build']`

Now, both of these run the *generate* target because they independently need to make sure all the code generator source is processed and available, but if we run the *release* target we mean to generate the documentation and build the software for a release.

You would think it makes sense for **script.py** to resolve the dependencies like this:

`['generate' 'idiopidae' 'latex' 'docs' 'compile' 'link' 'build']`

for the *release* target, but what I found is that many of these commands can be destructive or set options that create different output that chokes later targets in other dependencies. In the above example, what if we generate a parser file without debugging for documentation purposes in *docs*? Then when the resulting output reaches the *build* target it will actually be the wrong source, so we want to run *generate* again for build under the assumption that it changes.

This is exactly why file-centric builds don't work as well as target-centric, and why each list of dependent targets has to stand alone. Targets in later chains have to rely on their lists of targets being run in a consistent order, so if they are removed or screwed up in some way then you have brittle target chains. A file centric build is **not** going to rerun targets for later tasks because it assumes the target file is built correctly for **all** targets. The simple example above of debug vs. "clean" versions of a generated file are good examples.

Rather than force people to make lots of different targets for inside various chains, Vellum simply decides to be more verbose or explicit and keep all targets in a dependency chain, even if they've been run earlier. The only reliable exception is when a target is repeated immediately. In this case it's safe to remove it.

This also fits into why Vellum ignores targets it knows nothing about rather than report an error. It makes the build easier to write and use since it's harmless to just not do something, and there's enough logging to see that a target did nothing. In fact, the point of a build tool like Vellum is that you can run a single target on its own to see if it works. If you didn't need this ability to run a target out of context then you could just use a simple shell script and skip all this software.

That makes up the majority of *Script* except for some methods for showing it and the *__init__* method. You can read the code for those, but really you should be moving on to *Scribe* to see how the meat of the application actually runs.

A.5 Scribe

The *Scribe* class is where all the action happens for Vellum. It takes the results of the parsing, structuring, and organizing of the previous steps and then executes what it is told in the right order. In a way it's a poorly implemented virtual machine for Turing tar-pit language. This means you should be able to understand it easily unlike a real virtual machine.

Things start off like normal with the class definition and the initialization:

Code: scribe.py:class Scribe – *Getting It All Going*

```
8  class Scribe(object):
9      """
10     Turns a build spec into something that can actually run.  Scribe
11     is responsible for taking the results from Script and loading
12     the extra commands out of ~/.vellum/modules so that you can run
```

```
    it.

    This is the equiv. of the component in an interpreter that
    processes a cleaned and structured AST to execute it.
    """

    def __init__(self, script):
        self.script = script
        self.options = self.script.options
        self.target = None
        self.line = 1
        self.source = os.path.expanduser("~/.vellum/modules")
        self.stack = []
        self.commands = self.script.commands

```

There are then some tiny methods for getting options, logging messages, and dying on errors:

Code: scribe.py:support methods – *Misc. Support Gear*

```
    def option(self, name):
        """Tells if there's an option of this type."""
        return self.options.get(name,None)

    def log(self, msg):
        """
        Logs a message to the screen, but only if "verbose"
        option (not quiet).
        """
        if self.option("verbose"):
            print msg
            sys.stdout.flush()

    def die(self, cmd, msg=""):
        """
        Dies with an error message for the given command listing
        the target and line number in that target.
        """
        if not self.option("keep_going"):
            raise DieError(self.target, self.line, cmd, msg)

```

We then have a few methods that handle different aspects of the available targets. Remember that targets are just a dict off the `self.script` variable as `self.script.targets` which we have *Script* to thank for all it's work. All we need to do is look them up in this dict and deal with them.

Code: scribe.py:handling targets – *Is?, Body, and Parsing*

```
    def option(self, name):
        """Tells if there's an option of this type."""
```

```
30          return self.options.get(name,None)
31
32      def log(self, msg):
33          """
34          Logs a message to the screen, but only if "verbose"
35          option (not quiet).
36          """
37          if self.option("verbose"):
38              print msg
39              sys.stdout.flush()
40
41      def die(self, cmd, msg=""):
42          """
43          Dies with an error message for the given command listing
44          the target and line number in that target.
45          """
46          if not self.option("keep_going"):
47              raise DieError(self.target, self.line, cmd, msg)
48
```

The *body_of_target* is just a convenience method for grabbing what the target represents. After that *is_target* will safely tell *Scribe* if this is a valid target.

The real work of target management is in the *parse_target* where the target's body is analyzed to determine how to run it. One of the things Vellum does is allow you to give a target's specification in a few ways.

1. A single *string* or *line string*.
2. A single *Reference* to a command to run.
3. A *list* combining any of the previous two. [2]

The purpose of *parse_target* is to figure out via *isinstance* calls whether the body of a target is one of these three situations. The contract is that no matter what, *parse_target* will return an array of *basestring* and/or *Reference* objects.

Scribe must also manage commands that are allowed to operate during the later *execute* and *transition* phase of operation. This turns out to be very simple thanks to *Press* loading all the available commands as simple Python functions for us.

Code: scribe.py:handling commands – *Command Finding*

```
77      def is_command(self, name):
78          """Tells the scribe if this name is an actual command."""
79          return callable(self.commands.get(name, None))
80
81      def command(self, name, expr):
82          """
83          Runs the command for the given name.  Pulls it out of the
84          self.commands.  Normally it just passes expr to the
```

[2] All commands should also support these combinations when it makes sense, and specify when they don't.

```
85          command, but if expr is a dict then it will call it
86          with **expr so you can do simpler kword commands.  If
87          you don't want this then you just define your command as
88          taking **args.
89          """
90          try:
91              to_call = self.commands[name]
92          except KeyError, err:
93              self.die(name, "Invalid command name %s, use -C to find out what's available.")
94
95          if isinstance(expr, dict):
96              return to_call(self, **expr)
97          else:
98              return to_call(self, expr)
99
```

The *command* method is fairly simple, as it just blows up with a *die* call if you try to use a command that doesn't exist. I actually think this might be better done by having it default to a die command that when called blows up like this, but for now this is more direct.

What happens next is a bit of secret sauce that you will need to understand to find out how commands shown in Section A.6 work.

Remember from Section A.2 that a command is "invoked" using the *Reference* class, so it has a name and a generic *expr* attribute. This means that your commands can take just about anything, but the point of having dicts in Vellum syntax is so that commands can be called with named parameters in the same way Smalltalk does it. However, managing these keyword arguments in Python is best done using Python's built-in syntax.

This leads to the last two lines above that, if the *expr* is a dict, it expands the dict out with ***expr* so that the method is called keyword argument style. Take a look at A.6 and specifically the *forall* function for a really good example. They really just map right onto Python's existing syntax.

A.5.1 Target Execution

Targets are actually executed by three methods: *execute*, *transition* and *build*. The *execute* method does all the real work of incrementing line numbers and running each element of the list that *parse_target* returns.

Code: scribe.py:execute target body – *Execute Each Target "Line"*

```
101     def execute(self, body):
102         """
103         Executes the body which can be anything parse_target() can
104         handle.  It properly handles the difference between a plain
105         string (shell command(s)) or a Reference (do some command),
106         or a list of those two.  Assumes you call self.start_target()
107         """
108         for cmd in self.parse_target(body):
109             if "__builtins__" in self.options:
110                 self.die(cmd, "Your command leaked __builtins__."
111                          "Use scribe.push_scope and "
```

```
                        "scribe.pop_scope.")

            self.line += 1
            if isinstance(cmd, Reference):
                # Reference objects are indications to run some
                # Python command rather than a shell.
                if self.is_command(cmd.name):
                    # discontinue on True
                    if self.command(cmd.name, cmd.expr):
                        self.log("<-- %s" % cmd)
                        return
                else:
                    self.die(cmd,
                             "Invalid command reference, available "
                             "commands are:\n%r." %
                             sorted(self.commands.keys()))
            else:
                # it's just shell
                cmd = cmd.strip()
                if cmd: self.command("sh", cmd)
```

This is the most complex method in *Scribe* which determines if the line being run from a target's body is a *Reference* or a string. If it's a string then Vellum assumes that a shell command should run so *self.command("sh", cmd)* gets run to do that work. Otherwise it's a *Reference* which means to run some Python defined command function.

Code: scribe.py:transition to target – *Target Transitions*

```
    def transition(self, target):
        """
        The main engine of the whole thing, it will transition to
        the given target and then process it's commands listed.  It
        properly figures out if this is a command reference or a
        plain string to run as a shell.
        """
        if not self.is_target(target): return
        self.line = 0
        self.target = target
        body = self.body_of_target(target)
        self.execute(body)
```

Targets are executed by the above *transition* method, which does nothing more than combine calls to *is_target*, *body_of_target* and *execute* to make a target run in the right order with all the proper setup needed for *Scribe* to report line numbers.

Which leaves us with *build* doing nothing more than using *Script.resolve_targets* on the requested build list and looping through them to make things go via *transition*.

Code: scribe.py:running all targets – *Running The Full Target List*

```
146     def build(self, to_build):
147         """
148         Main entry point that resolves the main targets for
149         those listed in to_build and then runs the results in order.
150         """
151         building = self.script.resolve_targets(to_build)
152         self.log("BUILDING: %s" % building)
153         for target in building:
154             self.log("-->: %s" % target)
155             self.transition(target)
156
```

A.5.2 String Interpolation

Vellum considers each of the various strings a Python dict based string interpolation template. This means that if you have options with setting in them you can use those settings inside your shell commands and other strings like they're variables.

Code: scribe.py:string handling – *Lame Pseudo-Variables That Work*

```
157     def interpolate(self, cmd_name, expr):
158         """
159         Takes a string expression and interpolates it
160         using the self.options dict as the % paramter.
161         It prints more useful errors than you'd normally
162         get from Python.
163         """
164         err_name = "%s %r" % (cmd_name, expr)
165         try:
166             return expr % self.options
167         except ValueError, err:
168             self.die(err_name, "Expression has invalid format: %s" % err)
169         except KeyError, err:
170             self.die(err_name, "No key %s for format, available keys are: %r" % (err, sorted(self.op
171
```

These work great as a kind of pseudo-variable syntax that's familiar with anyone who knows Python's dict string syntax: `'hi: %(name)s' % {"name": "Zed}` for producing `'hi: Zed'`. You can then use the same syntax in your shell commands and many arguments.

A.6 commands.py

Vellum's commands are simply Python methods defined in **commands.py** which is self-documenting thanks to the documentation available via `vellum -C`. Rather than go through every command, I'll assume you can just read the code at this point and figure out what it contains and how each one works. I'll do a short discussion of the one command that is the most complex: *forall*.

Code: commands.py:forall – *The forall Command*

```
141 def forall(scribe, files=None, do=[], top=".", var="file"):
142     """
143     Iterates the commands in a do block over all the files
144     matching a given regex recursively.  You can put anything
145     you'd put in a normal target in the do block to be executed,
146     and when it is executed the 'var' variable is set to the
147     full path of each file.  This will also be in each task
148     you transition to with the 'needs' command.
149
150     forall assumes that you want the variable to be 'file'
151     so for most tasks you can leave that option out.  If you
152     nest forall expressions then you'll need to give each one
153     a different name (just like in a real language).
154
155     Usage: forall(files "*.py" var "file" do [ ... ])
156     """
157     scribe.log("forall: files %r top %r var %r" % (files, top, var))
158     if not files:
159         scribe.die("forall", "Must give a file matching "
160                    "pattern in parameter 'files'.")
161
162     matches = []
163     for path, dirs, fnames in os.walk(top):
164         paths = (os.path.join(path,f) for f in fnames)
165         matches.extend(fnmatch.filter(paths, files))
166
167     scribe.log("forall: matched %d files." % len(matches))
168     for f in matches:
169         scribe.push_scope({var: f, "files": matches, "var": var})
170         scribe.execute(do)
171         scribe.pop_scope()
172
```

The *forall* command takes a series of keyword arguments and then works on them to iterate a given block. Unlike other commands it expects a list of things to run, and then simply runs them on each file found. It uses *Scribe's* scoping methods to make sure that the called target bodies are not able to mess up the main *options* dict, and then it just calls *Scribe.execute*.

This is the most complex command, but hopefully you can understand it. Refer back to the previous sections to see what each of those methods does.

Another command to review the *cd* command as it shows how to properly scope a whole block of commands inside a chdir call.

Code: commands.py:cd – *The cd Command*

```
173 def cd(scribe, to=None, do=[]):
174     """
175     Temporarily changes to a given directory and then runs the
176     block in that directory.  When it's done it pops back to the
177     previous directory.
178     """
```

```
179     scribe.log(" cd: %s" % to)
180     to = scribe.interpolate("to", to)
181 
182     if not to:
183         scribe.die("cd", "Must specify to parameter to cd into.")
184     elif not os.path.exists(to):
185         scribe.die("cd", "Target chdir path '%s' does not exist." % to)
186 
187     curdir = os.path.abspath(os.path.curdir)
188     try:
189         os.chdir(to)
190         scribe.push_scope({"parent": curdir})
191         scribe.execute(do)
192         scribe.pop_scope()
193     finally:
194         os.chdir(curdir)
195 
```

Taking a look at this, it simply preserves the current directory, then attempts the code block inside a push/pop of the scope. Notice this block of code is close to the one in *forall* so it could be abstracted further. Also notice that it doesn't try to pop the scope off if there's an exception. This is on purpose so that the *options* dict at that moment can be inspected. It does do the chdir back though.

Appendix B

Converting From Make