Punk Rock Metaprogramming

Punk Rock Metaprogramming

DIY Code Generation in 5 Steps

by Jasmyn B. Weatherhead

(sample edition)

Introduction

I was exhausted. My body hurt all over, and my jaw had inexplicably developed a weird clicking as a side effect of the three day hackathon I had just endured. On top on the previous two weeks of frantic work to get the product, the most elaborate we had ever attempted, out before Halloween I was well past my breaking point, but we were finally at the finish line. Thanks, Adderall.

That's a thing I noticed about staying up for multiple days, things just start to break. That thing they told you about your body needing sleep to heal? That was very real. So my jaw was weird for no reason other than the deadline. And the code.

We had picked live wallpapers because they were faster to produce than games but sold for about the same. Every app was a lotto ticket and it was important to get as many tickets as possible, so development time was a paramount concern. Live wallpapers were a hot new feature, an Android exclusive, and I was smitten with them. It was 2012, judge me kindly. They were awesome, could be produced in a reasonable amount of time, and offered opportunity for artistic expression in a field that can be very dry. Also I had just had a minor hit with one of the first few I released to test the waters. The sales kept pouring in, and every time I heard my phone make the cha-ching! sound I knew I was on the right path.

So when the Great Recession hit and my partner and I were in need of a plan we knew what we had to do. We shipped product like our lives depended on it. Nineteen apps in just a few months. We were hungry, and this was our full-time job.

Little Witch was our twentieth app and it was very elaborate, with over a dozen touch effects that all had to be layed-out, animated, and coded, laboriously, by hand. Getting it done was a huge task, racing against the Halloween deadline so all our work wouldn't be for naught. Most nights we slept at our desks. It took everything we had but we released just on time and there was much rejoicing.

After it launched something inside me snapped. I had had enough. No way am I doing this again by hand when I knew there was a better way. There had to be. Not one more freaking time. In my heart I knew it could be different. Less of a burden. More of a joy. After coding so many line by line, learning the system and perfecting my form, the veil lifted and I was enlightened. I saw the underlying patterns, the ordo ab chao, and I knew what had to be done. Before I lost any more hit points. Before I broke something.

After I recovered, I began my research. I lost a couple days looking up popular metaprogramming frameworks. This just depressed me. The available frameworks all seemed so complicated, I felt I would spend all my time learning them and I'd be no better off than when I started.

Screw that.

So I set off on my own to find my own solution. Two weeks later I had an app machine. I had designed a custom language just for my problem space and written a parser and all the code needed to generate apps directly from the designer's mockup. It was wild.

Before this development time for every app took anywhere from two days to two weeks. Version 1.0 of my app machine, AutoLW, got this down to just a few hours. Version 2.0, several months

later, brought development time down to just half an hour. It takes as long now to post them at it takes to code them.

This changed everything. Soon we moved into our first office, a brick and mortar business in that one of the eleven walls of our single room was made up of the chimney from the Italian restaurant below. My productivity was through the roof, and even doing ten a day became commonplace. Not reskins. Not shovelware. Original scenes produced at an incredible pace by my art team, whose job I have yet to automate.

Within a year we had over 200 unique apps and several number ones. Today my company is in the top 1% of Play Store developers by app count, and the apps are work I'm proud of. We made a good income producing original animations and never resorted to ads. All of a sudden Intel and Hooli are calling me up! Wild stuff, and I owe it all to metaprogramming.

Step 0 – Overview

Metaprogramming has a couple of meanings. Here's the definition we'll be using in this book: Metaprogramming is writing code that writes code.

Metaprogramming is incredibly powerful, but it can seem too complicated to put into practice. Punk Rock Metaprogramming is a simple, effective approach you can start implementing now. It's really not that hard. Plus, it's fun. Working at a higher level of abstraction multiplies your power in a way that must experienced. When I'm metaprogramming it's like I can feel the power surging through my veins. It's addictive. These days when I'm only solving a single instance of a problem I get all twitchy. My antennae go off like it's code debt. It's that powerful.

I'll be showing you how to write your own custom Domain Specific Language (DSL) that will let you easily generate tons of unique code. This is high-end, exotic stuff, but we'll use an approach that makes it attainable for coders of all levels, with impressive results. Instead of learning esoteric new languages or frameworks and being swamped in jargon, we're going to take the punk rock approach and do it NOW, with whatever tools we have at hand. There's only one major new concept: learning to see things in terms of the highest abstraction possible. Once you have that I'll show you how to exploit your understanding with a simple, replicable process.

The truth is you don't need to learn any extra languages, frameworks, or metaprogramming IDEs. In fact you can use pretty much any language in which you find it easy or enjoyable to program. They can all parse strings and generate text which is really all we're doing on a mechanical level, and implementing these features is trivial. The real work comes before, during the abstraction phase.

The Process

Analyze > **Symbolize** > **Codify** > **Verify** > **Iterate**

Each of the steps, when taken by itself is very simple. Together they can be immensely powerful (and lucrative!) I will walk you through all the steps until you feel confident applying this extremely powerful technique to your projects. I know it may seem intimidating but it's really quite attainable. You just need a basic grasp of your chosen language and a deep understanding of the problem space.

Metaprogramming isn’t esoteric. It isn’t even that hard. The main shift is point of view. Changing our viewpoint to the highest level of abstraction possible is the fundamental shift required. The rest is easy.

Building a Code Machine

The ability to design and implement your own custom DSLs is very useful for all kinds of problem spaces, but it's especially good when you have to create lots of similar but unique codebases. App creation is an excellent example. Businesses often carry a range of different but similar apps in their stable, and generating more is crucial to keeping revenue streams high. In cases like these metaprogramming can be a life saver, freeing up countless hours previously spent in drudgery and

easily generating enough code to pay the bills. Metaprogramming is a great fit for anything where there’s variations on a theme.

Do not try to put the meta before the programming. That is to say you should have an existing codebase to work with. We will be writing a DSL that allows for easy generation of more instances of this type. You must have a target before you can reach it.

What is a DSL?

A DSL, or Domain Specific Language, is a small language custom made for a specific problem space. They aren't hard to write and can be written in any language; the challenge is effectively designing a DSL that fits your problem space. If you don't know the area backwards and forwards you are not yet ready to write your DSL. If you do have a thorough understanding of your problem space then congratulations. You are about to shock and amaze with your incredible new code generator. Try not to let it go to your head.

DSLs are, as mentioned above, small languages. They are designed to solve only one problem, and solve it well, but they are not general purpose languages like JavaScript or Ruby. However minimal they may appear the leverage a well designed DSL can give is enormous. You just need to start from a thorough understanding of your problem space.

The process is essentially to find the dynamic bits, assign them symbols then use these symbols as the syntax of your new language. You will also write a parser to convert (or transpile) your code into code that will run on your target system. This does not need to be the language that you wrote the transpiler in. For example All, our example DSL, is transpiled to Java code even though the transpiler is written in Ruby.

DIY Forever

This book will not focus on specific implementation syntax. This information goes rapidly out of date, and would limit your application of these principles unnecessarily. Instead we will focus on honing our understanding of this process for use in any situation. The truth is it does not matter what language you write your parser in, and the details of your implementations will shift over time. The real gem is the underlying technique which, though simple, can be sucessfully applied to any number of scenarios.

Example Project – AutoLW

To help make the process more concrete we will be working our way through the software I created while developing this process, AutoLW, a Live Wallpaper generator, and All, the associated DSL. Every situation is unique, but hopefully seeing how these principles are put into action will help you find ways to apply them to your situation.

Step 1 –Analyze

Study / Observation

Most of the work of metaprogramming is proper analysis of the problem space. The code you write is fairly simple. The magic is in the analysis. This step is all important. Take your time.

When generating multiple similar yet distinct codebases there will be code that changes (we'll call this dynamic) and code that stay the same (we'll call this static). The static bits are easy. Let's look at the dynamic.

The dynamic code is where the real meat is. You will find that often the dynamic code changes according to certain patterns, or perhaps certain combinatorials of options with various attributes. Your task at this stage is to identify the different patterns that the dynamic code takes. Identify its many forms, and see them for what they are. If you already have examples of what you're trying to generate, recommended if you're wanting to metacode it, study them for patterns. Stop thinking in terms of individual codebases, and learn to see each one as an example of a universal form. Tease out the shape of this form. Learn its variations and see the true form standing behind the specific example. This is the same basic skill you need for any high level programming, the same that lets you define classes in OOP, so you have the skills you need. You just need to internalize this new way of applying them.

Notebooks A'Plenty! (Docs start now)

Gather together all the exant codebase examples, and get a notebook. As you read through the code take notes of what's static and what's dynamic. These notes will be your prime resource when you go to symbolize your abstractions.

Analysis

You have to know the problem space as intimately as possible. Daily drudgery is a prime candidate. You have to know what's dynamic and what's static, and what variations are possible in order to effectively design your DSL. Study your codebase until the atomic elements of the dynamic code emerge.

You've spent time studying the extant codebase. All the specific instances of the hidden meta class. Now comes the time to organize what you've seen into a a coherent model. You must become intimately familiar with the processes and concepts in play so that the underlying abstractions will reveal themselves to you. Only when you know the problem space blindfolded in the dark are you ready to begin.

Basically at this stage you're looking for patterns. Abstraction is a tricky subject. It's both the most important aspect of the entire project and kind of ineffable. It's so vital and yet so difficult to teach. Once you've absorbed the extant codebases sufficiently the clouds should part, and you should begin to sort the dynamic from the static. More importantly you should begin to see the recurring

patterns, the members of the set of options, whose combinatorials are your finished product. Remember, you can always fix any errors, so don’t be too intimidated. Study the code until you think you have an angle, then dive in.

This has been a sample of

Punk Rock Metaprogramming – DIY Code Generation in 5 steps

Please visit

www.gum.co/punkrockmetaprogramming

to pre-order the full book.

Thanks for reading! :)