Timothy Boyarski
2011 Winter Cres, Coquitlam, BC V3K 6T6
COMP2990: January 2017 – August 2017

August 31st, 2017

Dear Susanna Khan,

The BC Cancer Agency (BCCA) is an academic institution dedicated to cancer research, prevention, and treatment. I worked under the within the BCCA; specifically, for the Lymphoid Cancer Research Department (LCR) of the BC Cancer Research Centre (BCCRC). The LCR is comprised mainly of wet-lab scientists, clinicians, and bioinformaticians. I was involved with the latter group and worked alongside one colleague: Fariha [BSc]. I was under the direct supervision of Lauren Chong [MSc] and Dr. Stacy Hung [PhD], and we were all lead by Principal Investigator Dr. Christian Steidl [MD, PhD]. The focus of the research group was to characterize and elucidate the genes and gene variants associated with various types of lymphoid cancer. The research was performed with the intention of advancing and improving clinical treatments and outcomes for lymphoma patients.

I began by investigating tools related to bioinformatic processing. The department had utilized GNU Make to automate the processing of experimental data by bioinformatic tools. Because of limitations mentioned in my interim report, we adopted a more versatile language (Snakemake) to automate our analysis pipelines. This required research into computer programming languages, workflow management systems, and of the analytical tools utilized by our team. A notable area of difficultly for this project has been the lack of available examples and communities; the tools we are utilizing are relatively new and offer limited documentation and support.

Overall, the work term provided an in-depth experience in the bioinformatics field. Working alongside wet-lab scientists, clinicians, and bioinformaticians has provided a comprehensive view of what a career in this field requires and what it would entail.

Regards,

Tim Boyarski, BSc

APPLICATION OF SOFTWARE DESIGN STRATEGIES DURING THE DEVELOPMENT OF MODULAR SNAKEMAKE PIPELINES

Tim Boyarski – A00979491

Computer Systems Technology

COMP 2990: Jan 2017 – Aug 2017

BC Cancer Research Centre

675 West 10th Avenue, Vancouver, BC, V5L 1L3

Supervised By: Lauren Chong

August 31st, 2017

Submitted To:

Susanna Khan, British Columbia Institute of Technology

Summary

The application of design principles resulted in a efficient, user-friendly, and maintainable system. The language of Snakemake and the design of the system were not identified as limiting factors; rather, the system is limited by the ability of the developer or the processing-power of the hardware. The application of design patterns was most notable when clearly defining system boundaries, establishing workspace-specific configuration, and automating workspace setup. This report discusses the utilization of software design patterns (Gamma, 1995; Larman, 2012) as they relate to specific sections of the LCR Snakemake Pipeline System.

Table of Contents

List of Figures

Introduction

Initially designed with the intent of automating the compilation of computing languages such as C, *make* was first introduced by Stu Feldman in 1979 for AT&T Eighth Edition Research Unix. Almost ten years later it was succeeded by the more efficient *mk*, designed by Andrew Hume of Bell Laboratories (Hume, 1987). In 1991, *GNU Make* was released (Stallman & McGrath, 1991), and it is still being actively maintained (Free Software Foundation, 2016).

It's a language that has been embraced by the scientific community in the hopes of streamlining the reproducibility of research (Schwab, Karrenbach, & Claerbout, 2000). GNU Make has since become an integral piece, especially within the bioinformatics community (Parker , Gorlick, & Lee, 2003).

The core functionality and scope of the 1991 version of *GNU Make* has been maintained throughout numerous versions. Certain caveats become apparent over time (Conifer Systems LLC, 2010; Miller, 1998). The caveats soon became limitations. As scientists began overcoming these limitations with ad-hoc code, the reproducibility and transparency of research diminished as a consequence. After realizing the complexity of our existing *GNU Make* system (Chan, Hung, & Chong, 2016), it became apparent that expansion was becoming difficult. Due to the existing system's immense coupling and lack of cohesion, robust scaling and integration of new features was becoming problematic.

The in-house *GNU Make* system provided a foundation upon which a new system would be based, and this greatly accelerated the design process. It was important to understand the existing system, why it was valued, and what it was missing. The *GNU Make* system was analyzed for its benefits and drawbacks. The benefits were to be further supported, and the drawbacks ideally eliminated. With an existing system implemented, it was also much easier to realize system wide patterns. This provided immense value when abstracting, isolating, and compartmentalizing components in the new system.

The new organizational and structural requirements of processing the vast amounts of data produced by Next Generation Sequencing (NGS) could not be sufficiently met by *GNU Make*; as a result, scientists began searching for and developing build automation alternatives. The emerging languages intended to improve upon *GNU Make* by providing additional features such as high-performance computing (HPC) cluster and environmental management integration, better logging, granular configuration, and better error recovery. Our department began an exploration into alternative workflow tools such as *BigDataScript* (Cingolani, Sladek, & Blanchette, BigDataScript: a scripting language for data pipelines, 2014), *Nextflow* (Di Tommaso, Chatzou, Baraja, & Notredame, 2014), and *Snakemake* (Köster & Rahmann, 2012). There is much debate about the advantages and drawbacks of the various build automation languages made accessible to the public (Leipzig, 2017).

It has since been determined that *Snakemake* is the best language to replace the existing *GNU Make* system. This article focuses solely on design applications for build automation within Snakemake; however, that is not to say the ideas discussed here are exclusive to design within Snakemake.

Design Patterns and Recommendations

The existence, understanding, and leveraging of design patterns is common place in civil engineering (Alexander, 1977). The value which patterns provide to civil engineers is obvious. Bridges, boats, and planes provide transportation across rivers, lakes, and oceans. The value of design patterns to software engineers is not as easily understood or as quickly realized.

Software patterns are abstract solutions to specific problems. They act as guidelines when making design decisions. They are tried and tested design solutions to commonly occurring design problems. They are well understood by software developers and architects, and aid by providing meaningful instructions and constraints during the design process. A common reason for application of software patterns being, "…if the solution is easier to comprehend, then by extension, it will also be easier to maintain" (Burford, 2014). Additionally, others argue for their value at a higher level, "the critical design tool for software development is a mind well educated in design principles" (Larman, 2012).

By having a descriptive and referenceable design framework, users are better able to understand of the role, functionality, and position of the system-feature or system-element they are designing. With a more informed user, a user with a better understanding of the code to be implemented, there is an increased likelihood that the code will be implemented in the correct position, given a proper amount of generalization, and that it falls consistently within the decision-making processes used to resolve other organizational conflicts within the system. By providing structure to the resolution of design conflicts, we are able to increase system consistency as users will be more likely to provide the same solution to the same question, even if it's in a different section of the program.

Software patterns can be represented as stand-alone ideas; however, they are typically provided as a set (Wolfgang, 1994; Gamma, 1995). They are often discussed within the realm of object-oriented programming, and UML diagrams (Sunyé, Le Guennec, & Jézéquel, 2012; Larman, 2012). Others, including myself, believe that software patterns can be applied to more than just object oriented languages (Duell, Goodsen, & Rising, 1997).

Python (Python Software Foundation, 2017), is a true object-oriented language (Tutorialspoint.com, 2017). The object-oriented designation of Snakemake is potentially debatable. Rules can be thought of as classes, and directives their attributes. We execute rules in the same way a user would instantiate an object. Notably, as Snakemake inherits and is built in Python, we will otherwise consider the entire system to be eligible for the application of object-oriented design.

For the purpose of code usability, maintenance, and understanding, at least within the realm of scientific research, good software design must also consider reproducibility, and tracking provenance. Below are a few more recommendations of note (Kanwal, Khan, Lonie, & Sinnott, 2017).

> *"Workflow developers should avoid hardcoding environmental parameters such as file names, absolute file paths and directory names that would otherwise render their workflow dependent on a specific environment setup and configuration."*

> *"Workflow developers should provide a complete data flow diagram serving as a blue print containing all the artefacts including tools, input data, intermediate data products, supporting resources, processes and the connection between these artefacts."*

> *"Tools should either be packaged along with the workflow or made available via public repositories to ensure accessibility to the exact same versions and parameter settings as used in the analysis being reproduced, hence supporting flexible and customizable workflows."*

"This factor might be considered out of control of the workflow developers but detailed documentation of the underlying framework used and community support can help in overcoming the associated learning curve."

Other considerations include:

- Simplicity: An extensive setup process detracts from a researcher's ability to quickly and easily try out new ideas. Using supportive scripts allows for the black-boxing of, and greatly speeds up, the pipeline setup process.
- User awareness: Configuration variables are most likely to be set if the user is aware that they exist. By centralizing configuration, we can increase user awareness which in turn increases their ability to customize and fully utilize the pipeline.
- Shell call logging: If CLI over-rides are used, it is likely they will not be captured in the systems' logging. With respect to auditing, there is immense value in saving the exact shell call executed. Manual tracing of code introduces human error. Systems must be able to log shell calls in a dry-run and in real-time.
- Robust versioning: Often different versions of the same bioinformatics tools are used. For reproducibility, and for auditing purposes, it is important to be able to keep, replicate, or revert to various versions of the same tool.

Readers must also be made aware that design considerations for the code base were heavily influenced by the intended user audience. Their above average skill-level permitted for aggressive abstraction and it encouraged the integration of python-based convenience scripting. This paper will demonstrate the application of software patterns in the development of a Snakemake system focused on generating build automation pipelines for bioinformatic purposes. The system was designed for the Lymphoid Cancer Research Department of the BC Cancer Agency.

Author's Background

I have been an avid computer user my entire life. I was first introduced to scripting when I started running a network of bots to play (farm) video games for me. I was formally introduced to coding at the University of Victoria. While earning a BSc in Biology, I took electives in computer science and software engineering.

I've recently completed the first year of the Computer Systems Technology (CST) Diploma at British Columbia Institute of Technology (BCIT). Here I learned about web-development, and extended my knowledge in databases, C, Java, and Python. During this Co-op position, I've continued to take program-relevant courses part-time. I began a part-time course in Object Oriented Analysis and Design (OOAD) at BCIT from January-April of 2017. This course provided the foundational design pattern knowledge which I am using to write this paper; specifically, it was within this course that I was introduced to Larman's General Responsibility Assignment Software Patterns (GRASP).

Methodology

The underlying Snakemake system to which software design-patterns are applied was developed during an 8-month Co-op position with the British Columbia Cancer Agency (BCCA) with its Lymphoid Cancer Research Department (LCR). The goal of the position was to convert and improve upon an existing system, written in GNU Make. The design process began after careful consideration of the language to be used. As a group, we understood that our choice needed to take into account the computer literacy of the user base, and the budget available. Leipzig (2017) has written an excellent review of the current landscape.

Language Considerations

Nextflow (Di Tommaso, Chatzou, Baraja, & Notredame, 2014): The language significantly empowers the coder; however, it also requires a user with a significant computer science background. Even though the language was built on Java and Bash, it introduced a formidable amount of novel terminology and formatting. Furthermore, the developers decided to heavily rely on a programming paradigm not commonly taught in academic institutions (Guo, 2014). The languages taught by institutions, which include Python, C, C++ and Java, comprise four of the five most-used languages globally (TIOBE software BV, 2017). Although increasing in popularity, languages like Swift and Go rank 11th and 16th, respectively. The difference between the two paradigms being a stream of data, versus a stream in control instructions. In addition to novel syntax and implementation strategies, the adoption of a paradigm likely to be foreign to new users greatly steepens the learning curve.

BigDataScript (Cingolani, Sladek, & Blanchette, 2014): It similarly uses methods, functions and classes, like Java and C; however, it uniquely provides a dependency operator by which modules can be linked. Because it more closely relates the design paradigms of the most commonly used languages (TIOBE software BV, 2017), this would have been our second choice had our group not already had a strong familiarity with GNU Make. The author of this language is the author of *snpEff* (Cingolani, et al., 2012).

Snakemake (Köster & Rahmann, 2012)**:** Snakemake combines the build automation functionality of GNU Make, with the simplistic and commonly taught language Python. Notably, Python is often the first language to be taught to new computer scientists (Donaldson, 2017; Fangohr, 2004). Snakemake addressed and provided functionality which was lacking in GNU Make like cluster support, control flow, inflexible wildcards, and minimal reporting mechanisms (Koster, 2014).

Production Environment and Hardware

The system was developed on Canada's Michael Smith Genome Sciences Science (GSC) HPC cluster, which runs on separate servers RedHat CentOS5 and RedHat CentOS6 operating systems. The server's head node was accessed via SSH from an iMac (Retina 5K, 27-inch). The five head-nodes contain between 8-32 CPUs and possess between 47-396GB of memory. Computational processing is delegated from a head node to at least one of the approximately 500 child-nodes. The child-nodes contain 24 CPUs and 47GB of memory each. Parallel processing was limited to a maximum of 100 child-nodes. Our department has currently been allotted approximately 20TB of disk space. The maintenance and upkeep of the servers was performed exclusively by the GSC.

Results

New definitions for system nomenclature are used to describe the workspace organization of the resultant system, and its reliance on supporting Python and Snakemake modules.

System Nomenclature

Module: These are considered to be the highest-level of coupling and compartmentalization. They are used to group sub-modules on the basis of their purpose and the types of files they process. Core process categories within this system have been coined and are as follows:

- Gen – The generation of files.
- Util – Utilities providing the ability to manipulate existing file formats.
- Annotate – Programs able to annotate existing file formats
- Metrics – Programs able to generate metrics on existing file formats.

Every module will contain an “INCLUDE” file, a python script, and one or more sub-modules:

- “moduleName_INCLUDE” – This file is responsible for the control-flow within a module and contains information regarding internal submodule interactions and dependencies.
- “moduleName.py” – This file is responsible for supporting the sub-modules within this module. It writes module specific information to system’s YAML, JSON, and Snakefile.
- “submodule#” – Each submodule file contains a single Snakemake rule to produce an output from a given input, designed such that system coupling remains low.

The modular part of the system can be broken down into two distinct categories of module, Python Modules and Snakemake Modules. The two categories are responsible for the setup and the execution of a pipeline, respectively.

Workspace Organization

The core structure for a workspace in this system can be generalized. It will always contain a Snakefile and corresponding configuration files. The purpose of each of these files is explained in detail below:

buildPipe.py: This script creates and writes to the YAML (input/config.yaml) and JSON (input/config.yaml) configuration files, as well as the project specific Snakemake file (Snakefile). The information it writes is generated via its interaction with the supporting module-specific python scripts. A hypothetical buildPipe.py execution is represented in the sequence diagram Figure 1.

Snakefile: This file is the driver of the pipeline. It determines which modules are to be included, and it is here that the desired outputs are listed. The linking of the pipeline configuration file ("input/config.yaml") is done within this file, and the linking of the cluster configuration file is done when calling to execute this file. Furthermore, each module contained within also provides a list of the sub-module specific outputs generated. This allows users to specify which outputs they want to produce without directly interacting with sub-modules.

input/config.yaml: This file contains the system's pipeline specific configuration variables. Module specific python support scripts write the configurable and mandatory run-time parameters for every rule contained within the module. This information is read by Snakemake at the time of execution for each rule.

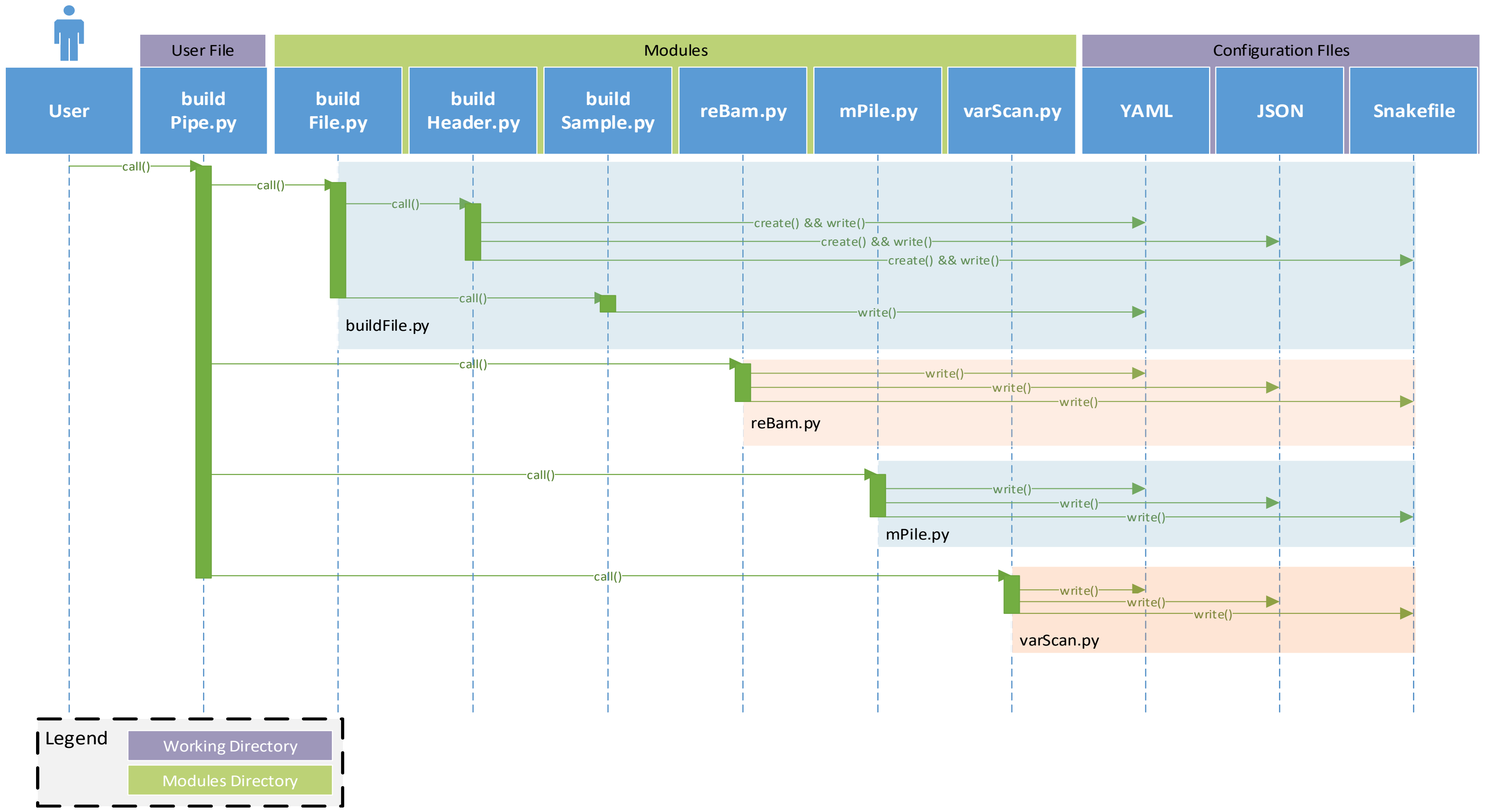


Figure 1. Visio sequence diagram of the user interaction with buildPipe.py. The diagram demonstrates the interactions amongst files in the build process of a the Snakemake pipeline. Note the modularity and low coupling of the system. The user directly interacts with only a single file. Each file that buildPipe.py interacts with is given the ability to direct write information to the three configuration files: YAML, JSON, and the Snakefile. Message labels are not factual method calls, rather they are meant to describe the actions being performed. As indicated by the legend, light purple background shading at the top of the diagram indicates the file would be within the current working directory. Light green background shading indicates the file is in the shared modules directory.

input/config.json: This file is to contain the system's cluster specific configuration variables. This information is read by Snakemake at the time of execution for each rule. The parameters are mainly used for hardware resource requests, job naming, and logging.

input/sampleFILE.txt: This metadata file is used to store the names of the samples to be processed by our pipeline. This is to allow the processing of a subset of files within a directory, and also allows for specific tracking and pairing of named tumor-normal samples. Due to the required pairing of tumors to normal, single and tumor-normal sample files are formatted differently. The sample names file may take two organizational structures, paired or single. Paired samples consist of two samples which reflect the tumor-normal biopsy of a single patient. In the later stages of a paired sample pipeline, the normal is considered to be a control for the tumor sample. Each tumor will have its own specific normal sample, as such, special attention must be paid to ensure that the tumor and normal samples are properly paired to each other. The proper pairing is accomplished by writing the tumor-normal pairs to a file with a header line to establish a column of tumor samples and a corresponding normal sample. Single samples are listed in a single sample file absent of a header.

input/samplePartList.txt: The sample part-list file is used when an input sample is provided in multiple parts, and is referenced when merging the samples together. Please note, the user is responsible for maintaining the sample integrity, there are no checks in place to prevent the merging of unrelated samples, samples are merged exactly as grouped in this file.

input/shellcalls.txt: Metadata file containing time-stamped pipeline shell calls.

input/runStats.txt: Metadata file containing timing metrics for the execution of each pipeline rule.

Python Modules

The Python modules are merely helper functions for the system. They are at no point involved in the functionality of the pipeline; rather, they are responsible for setting up the pipeline. The python modules are considered to be ‘features of convenience’, and are provided to improve user experience. There are two Python modules, and they are interacted with by the user in a consistent order. Users will always first interact with py_startHERE, and then they will interact with py_buildFILE. The purpose of these python modules is explained below:

py_startHERE: To ensure consistency across workspaces which utilize this Snakemake system, the module was created to automate the process of creating the workspace. Users are able to provide the absolute location, the name of the workspace, and the modules which are to be included in the workspace. Contained within the resultant workspace are the base files required for assembling a Snakemake pipeline from this system. Inside the directory are mock files for inputs, the listing of samples, the listing of sample parts, and a buildPipe.py file set up to interact with the requested modules. All of this was initially performed manually; however, due to the repetitive nature of this process and the expanding amount of information that needed to be transferred, the entire process is now automated.

py_buildFILE: Automating the pipeline setup became increasingly important as the number of configurable variables grew. A pipeline of any meaningful purpose will have at least a hundred different configurable variables. It became apparent that users could not be reasonably expected to list all these variables. It was determined that providing the user with access to all Snakemake system variables, which relate to configuration of the system, was of great convenience and value to the user. Absent of a script to accomplish this, the user would have otherwise manually navigated to each module only then to have to identify and correctly list shared and sub-module specific configuration variables. Scripting the whole process has also increased formatting consistency. Formatting consistency provides benefit when attempting to read or understand the configuration files of someone using the same system. This module is called by the supporting python script buildPIPE.py, which was built by the previous python module py_startHERE. Figure 1 aids in explaining the purpose of this module in the pipeline setup process.

Snakemake Modules

The Snakemake modules provide the system it's functionality. The output of one module is to be connected to the input of another. They are to be added and removed as necessary as to assemble a meaningful data-processing pipeline. These modules and their code are heavily based upon similar modules provided by the existing *GNU Make* system.

The modules are not agnostic to the software versions; however, they can be tweaked to use any version of the following software. The following software was controlled by the environmental manager Conda (Continuum Analytics, Inc, 2017) :

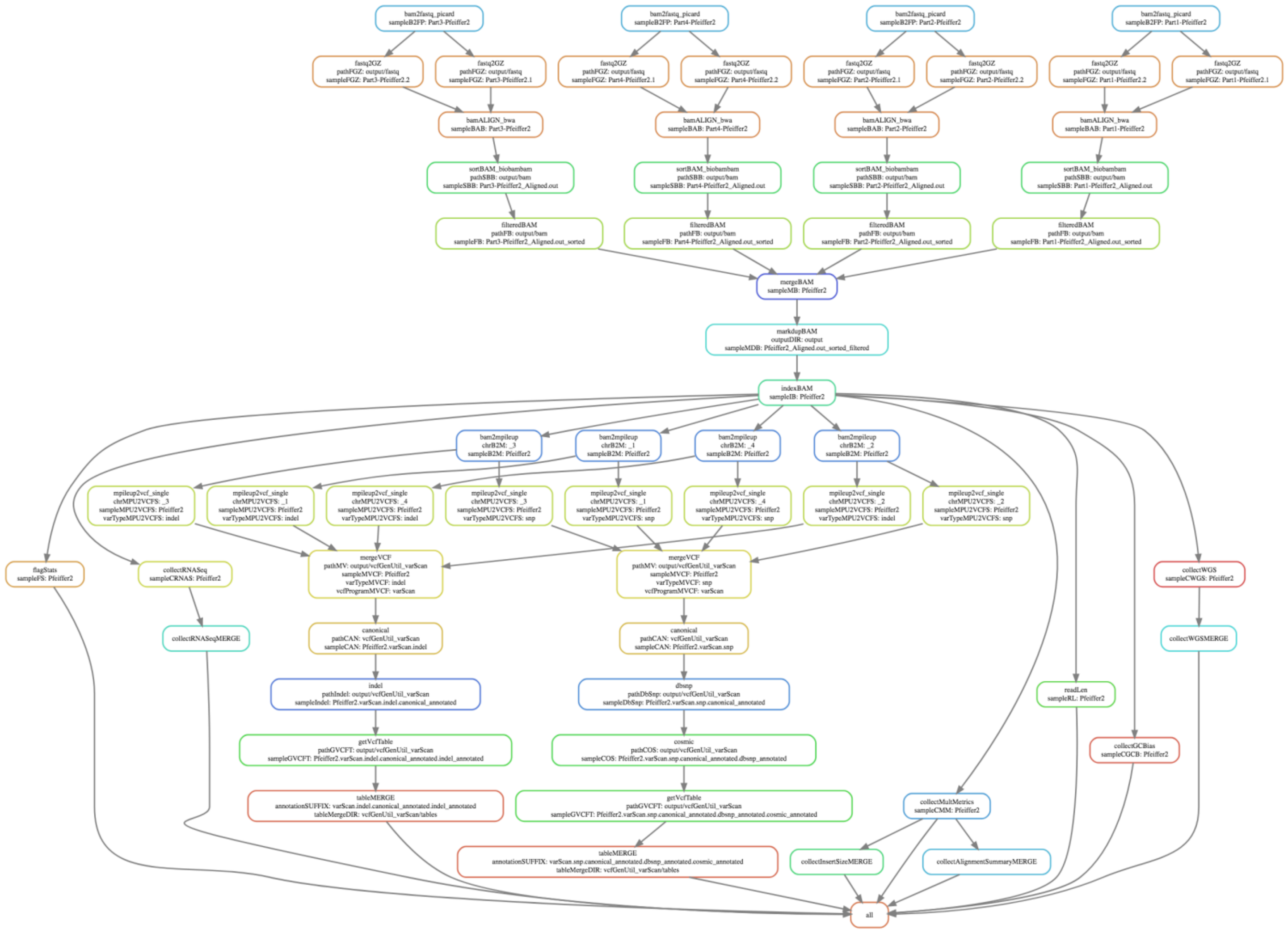
- VarScan (Koboldt, et al., 2012)
- Samtools (Li, et al., 2009)
- Bwa (Li & Durbin, 2010)
- Star (Dobin, et al., 2013)
- snpEff (Cingolani, et al., 2012)
- snpSift (Cingolani, et al., 2012)
- Picard (Broad Institute, 2017)

The names of the modules, and theory behind their organization is explained below:

- **fastqGen**: Sub-modules able to generate a '.fastq' file, from a non '.fastq' input.
- **fastqUtil**: Sub-modules able to manipulate, or post-process a file of format '.fastq'
- **bamGen**: Sub-modules able to generate a '.bam' file, from a non '.bam' input.
- **bamUtil**: Sub-modules able to manipulate, or post-process a file of format '.fastq'
- **mpileupGen**: Sub-modules able to generate a '.mpileup' file, from a non '.mpileup' input.
- **vcfGenUtil_varScan**: VarScan based sub-modules able to generate a file in '.vcf' format, from a non '.vcf' input source; as well, VarScan based submodules able to manipulate, or post-process a file produced by VarScan and of '.vcf' format.
- **vcfUtil**: Sub-modules able to manipulate, or post-process a file of format '.vcf'
- **bamMetrics**: Sub-modules able to generate analysis metrics for a '.bam' file.
- **genericUtil**: Sub-modules able to manipulate, or post-process a file of format '.txt'
- **vcfAnnotate**: Sub-modules able to annotate a file of format '.vcf'
- **starFusion**: Sub-module able to generate fusion metrics from a specialized '.junctions' file.

Final System

To understand the design considerations being suggested, users should be made aware of a finalized system which demonstrates their application. In the form of a directed acyclic graph (DAG), Figure 2 provides a visual representation of the resultant system, for which the code base can be found here: https://github.com/tboyarski/BCCRC-Snakemake



bam2fastq_picard sampleB2FP: Part3-Pfeiffer2
bam2fastq_picard sampleB2FP: Part4-Pfeiffer2
bam2fastq_picard sampleB2FP: Part2-Pfeiffer2
bam2fastq_picard sampleB2FP: Part1-Pfeiffer2
fastq2GZ pathFGZ: output/fastq sampleFGZ: Part3-Pfeiffer2.2
fastq2GZ pathFGZ: output/fastq sampleFGZ: Part3-Pfeiffer2.1
fastq2GZ pathFGZ: output/fastq sampleFGZ: Part4-Pfeiffer2.1
fastq2GZ pathFGZ: output/fastq sampleFGZ: Part4-Pfeiffer2.2
fastq2GZ pathFGZ: output/fastq sampleFGZ: Part2-Pfeiffer2.1
fastq2GZ pathFGZ: output/fastq sampleFGZ: Part2-Pfeiffer2.2
fastq2GZ pathFGZ: output/fastq sampleFGZ: Part1-Pfeiffer2.2
fastq2GZ pathFGZ: output/fastq sampleFGZ: Part1-Pfeiffer2.1
bamALIGN_bwa sampleBAB: Part3-Pfeiffer2
bamALIGN_bwa sampleBAB: Part4-Pfeiffer2
bamALIGN_bwa sampleBAB: Part2-Pfeiffer2
bamALIGN_bwa sampleBAB: Part1-Pfeiffer2
sortBAM_biobambam pathSBB: output/bam sampleSBB: Part3-Pfeiffer2_Aligned.out
sortBAM_biobambam pathSBB: output/bam sampleSBB: Part4-Pfeiffer2_Aligned.out
sortBAM_biobambam pathSBB: output/bam sampleSBB: Part2-Pfeiffer2_Aligned.out
sortBAM_biobambam pathSBB: output/bam sampleSBB: Part1-Pfeiffer2_Aligned.out
filteredBAM pathFB: output/bam sampleFB: Part3-Pfeiffer2_Aligned.out_sorted
filteredBAM pathFB: output/bam sampleFB: Part4-Pfeiffer2_Aligned.out_sorted
filteredBAM pathFB: output/bam sampleFB: Part2-Pfeiffer2_Aligned.out_sorted
filteredBAM pathFB: output/bam sampleFB: Part1-Pfeiffer2_Aligned.out_sorted
mergeBAM sampleMB: Pfeiffer2
markdupBAM outputDIR: output sampleMDB: Pfeiffer2_Aligned.out_sorted_filtered
indexBAM sampleIB: Pfeiffer2
bam2mpileup chrB2M: _3 sampleB2M: Pfeiffer2
bam2mpileup chrB2M: _1 sampleB2M: Pfeiffer2
bam2mpileup chrB2M: _4 sampleB2M: Pfeiffer2
bam2mpileup chrB2M: _2 sampleB2M: Pfeiffer2
mpileup2vcf_single chrMPU2VCFS: _3 sampleMPU2VCFS: Pfeiffer2 varTypeMPU2VCFS: indel
mpileup2vcf_single chrMPU2VCFS: _1 sampleMPU2VCFS: Pfeiffer2 varTypeMPU2VCFS: indel
mpileup2vcf_single chrMPU2VCFS: _4 sampleMPU2VCFS: Pfeiffer2 varTypeMPU2VCFS: indel
mpileup2vcf_single chrMPU2VCFS: _3 sampleMPU2VCFS: Pfeiffer2 varTypeMPU2VCFS: snp
mpileup2vcf_single chrMPU2VCFS: _1 sampleMPU2VCFS: Pfeiffer2 varTypeMPU2VCFS: snp
mpileup2vcf_single chrMPU2VCFS: _4 sampleMPU2VCFS: Pfeiffer2 varTypeMPU2VCFS: snp
mpileup2vcf_single chrMPU2VCFS: _2 sampleMPU2VCFS: Pfeiffer2 varTypeMPU2VCFS: indel
mpileup2vcf_single chrMPU2VCFS: _2 sampleMPU2VCFS: Pfeiffer2 varTypeMPU2VCFS: snp
flagStats sampleFS: Pfeiffer2
collectRNASeq sampleCRNAS: Pfeiffer2
mergeVCF pathMV: output/vcfGenUtil_varScan sampleMVCF: Pfeiffer2 varTypeMVCF: indel vcfProgramMVCF: varScan
mergeVCF pathMV: output/vcfGenUtil_varScan sampleMVCF: Pfeiffer2 varTypeMVCF: snp vcfProgramMVCF: varScan
collectWGS sampleCWGS: Pfeiffer2
collectRNASeqMERGE
canonical pathCAN: vcfGenUtil_varScan sampleCAN: Pfeiffer2.varScan.indel
canonical pathCAN: vcfGenUtil_varScan sampleCAN: Pfeiffer2.varScan.snp
collectWGSMERGE
indel pathIndel: output/vcfGenUtil_varScan sampleIndel: Pfeiffer2.varScan.indel.canonical_annotated
dbsnp pathDbSnp: output/vcfGenUtil_varScan sampleDbSnp: Pfeiffer2.varScan.snp.canonical_annotated
readLen sampleRL: Pfeiffer2
getVcfTable pathGVCFT: output/vcfGenUtil_varScan sampleGVCFT: Pfeiffer2.varScan.indel.canonical_annotated.indel_annotated
cosmic pathCOS: output/vcfGenUtil_varScan sampleCOS: Pfeiffer2.varScan.snp.canonical_annotated.dbsnp_annotated
collectGCBias sampleCGCB: Pfeiffer2
tableMERGE annotationSUFFIX: varScan.indel.canonical_annotated.indel_annotated tableMergeDIR: vcfGenUtil_varScan/tables
getVcfTable pathGVCFT: output/vcfGenUtil_varScan sampleGVCFT: Pfeiffer2.varScan.snp.canonical_annotated.dbsnp_annotated.cosmic_annotated
collectMultMetrics sampleCMM: Pfeiffer2
tableMERGE annotationSUFFIX: varScan.snp.canonical_annotated.dbsnp_annotated.cosmic_annotated tableMergeDIR: vcfGenUtil_varScan/tables
collectInsertSizeMERGE
collectAlignmentSummaryMERGE
all

Figure 2. Directed acyclic graphical (DAG) generate by Snakemake to represent largest possible build-automation pipeline for the Snakemake system. The example is processing a single sample, split in parts (4x).

Discussion

We will begin by demonstrating and discussing the application of design patterns by Larman. These patterns are identifiable as they are followed by a supporting question. Follows are the application of Gamma's patterns, after which we will discuss recommendations on reproducibility, and tracking provenance as previously describe by Kanwal et al. Lastly, personal considerations which fall outside of any specific design pattern will be discussed.

Patterns (Larman, 2012)

High cohesion – In which class is it best to assign the responsibility?

Assign a responsibility so that the cohesion remains high, keeping related things together.

- Pipeline specific configuration variables are grouped in a single location (input/config.yaml)
- Cluster specific configuration variables are grouped in a single location (input/config.json)
- Shell calls are time-stamped and are grouped in a single location (input/shellCalls.txt)
- Rule run-time metrics are calculated and stored in a single location (input/runStats.txt)
- Log files are first grouped by module, and then by sub-module.
- Output files cohesion is moderately low, they are generally grouped by file type.
- Modules are grouped first by the file format they operate on, and then by their purpose.

Protected variations – Designing when parts of our application are subject to predictable change?

Users must isolate the code subject to predictable change, via indirection.

- All the configuration parameters were put into a single file, that is user generated AND project specific. The project space can be easily customized to fit a specific user's needs.

Pure fabrication – Designing to implement a method for which there is no proper class for it?

Create/fabricate the class for the method.

- Created py_startHERE module to provide user convenient functions to generate a workspace.
- Created py_buildFILE module to provide user convenient functions to generate their pipeline.

Low coupling – In which class is it best to assign the responsibility?

Assign a responsibility so that the coupling remains low, limiting the number of new relationships.

- Ensure that 'ruleorder' is used in the "_INCLUDE" file, as this will allow it to rely on existing coupling within the system. Ideally, 'ruleorder' is ordering internal sub-modules, in doing so it does increase system coupling. System coupling is increased when using 'ruleorder' to relate to external rules; this should be avoided where ever possible.
- buildPipe.py does not interact with individual submodules, it interacts with the supporting module specific python scripts to provide the required variables for an entire module.

Information expert – In which class is it best to assign the responsibility?

Ensure responsibilities are given to the object which knows the most about what is going on.

- Module specific python supporting scripts are responsible for writing all module specific information to the Snakemake file, and the pipeline and cluster specific configuration files.
- The Snakefile has access to all of the configuration variables (via configfile) and all the rules (via include) of the system, as such, it is responsible for determining the output generated by the pipeline, and which rules and configuration variables are to be required.

Polymorphism – Designing when we have polymorphic methods?

Create a structured guideline or template, have all classes follow this structure.

- By naming each module's python helper function to be the exact same as the name of the module, we know we can access any module's helper function, without stepping into the module to find it.

Controller – How to de-couple the UI from the data layer?

Create an intermediary class, which delegates the events from the UI.

- Snakefile is the controller of the system. The system variables have been abstracted from it. System variables are all stored in pipeline (input/config.yaml) and cluster (input/config.json) specific configuration files.

Indirection – How can portions of a system be decoupled?

Create an intermediary class, or layer of classes, to provide separation, to decouple a system.

i. Controllers: Delegators
   - Snakefile indirectly interacts with the sub-modules controller, which is an '_INCLUDE' file. The controller is responsible for determining which sub-modules are included.

ii. Adaptors: Translators
   - buildPIPE.py indirectly interacts with sub-modules' translators, which are the '.py' files. The module specific translators provide a means for all sub-modules to write their required information to the YAML, a JSON, and Snakemake file.

Patterns (Gamma, 1995)

Abstract Factory

Provide an interface for creating families of related objects without specifying concrete classes.

- All Snakemake modules are supported by all the python modules.

The Builder

Abstract the construction of a complex object from its representation, so the same construction process can create different representations.

- The same python supporting template script is used by every sub-module to automate the writing to the YAML, JSON, and Snakefile.
- py_startHERE and py_buildFILE modules will be used to create every user's workspace.

Flyweight

Use sharing to support large numbers of objects efficiently

- By making the modules generic and under a single repository, all users in the code base are able to execute (instantiate) as many instances of the rule as they desire.

Mediator

Defines an object that controls how a set of objects interact. Loose coupling between colleague objects is achieved by having colleagues communicate with the Mediator, rather than one another

- The "_INCLDUE" files act as mediators, they control how the set of underlying sub-modules interact. Changes in constraints are localized to a single place for easier maintenance.

Momento

Captures and externalizes an object's internal state, so the object can be restored to that state later.

- The logging of the shell call prior to its execution.
- All configuration variables are stored in and read from file.
- Snakemake provides execution tracking and error recovery to allow for the continuation, or restarting of a pipeline from the step which was last successfully completed.

Observer

Defines a one to many relationship, so that when one object changes state, the others are notified and updated automatically.

- This service is being provided by Snakemake. If an upstream file is modified, everything downstream will be automatically reprocessed.

State

Pattern allows an object to change its behavior when its internal state changes.

- Any combination of modules can be requested from py_startHERE and py_buildFILE.
- Snakemake's Snakefile can accept any submodule specific output request.
- Upon system failure, incomplete files are deleted upon exit.
- Able to recovery from failure by starting from the last successfully processed file.

Recommendations (Kanwal, Khan, Lonie, & Sinnott, 2017)

Notably, from the recommendations previously listed by (Kanwal, Khan, Lonie, & Sinnott, 2017), the first coincides with the design pattern of protected variation. By isolating the hard-coding of absolute paths for genomic references, software binaries, and other configuration variables, we are better able to customize the behaviour of our pipeline as the changes we make will all be in a single file.

The second recommendation involved the visualization of the pipeline, in a graphical image format, as to increase user understanding and digestion of the pipeline. The language of Snakemake is able seamlessly provide this, as it is already a constituent component with the process of determining the pipeline's dependency structure. The visualization of a pipeline is made possible by piping the pipeline's dependency log into a program capable of visualizing it as a DAG (Gansner & North, 2000).

The third recommendation relates to reproducibility and transparency, it suggests that users make their entire system, and it's supporting components publicly available. Typically, this can be done by hosting the code base on a public repository; however, Snakemake provides support beyond this in two ways.

1) Support of the environment management tool *conda* (Continuum Analytics, Inc, 2017).
2) The ability to tarball the entire pipeline system for the purpose of compression during transportation via: "snakemake --archive my-workflow.tar.gz".

The last recommendation is that the intended audience be supported with proper documentation and community support. This is essential for any system that is to be expanded, users must be able to understand the existing code base and they need to be provided support in new areas. The Snakemake system has been extensively documented to provide a README documentation file for each sub-module. In addition, vignettes were produced to document the creation of pipelines and modules, such that the system can be readily expanded in the future.

Personal Considerations

With these recommendations in mind, and the previous design patterns in consideration, I realized that the intended audience was the most important consideration. Hastening the setup process, explicitly informing users of their configuration variables, providing better logging for auditing and bug tracking, and ensuring external reproducibility in research were all vitally important to a system which is well-designed and willfully adopted by its users.

Conclusion

The application of design principles during the development of the Snakemake system resulted in a loosely coupled, highly cohesive system with no immediately foreseeable technical limitations. The language of Snakemake and the design of the system were not identified as limiting factors; rather, the system is limited by the implementation ability of the module or pipeline developer and the computational power of the hardware on which the system is being run. As a result of good design, the system allows for adoption or extension by novel users. The simplicity, organization, and documentation of the system permits users lacking orientation in computer science access to its functionality. Snakemake is well-suited to exemplify design principles such as black-boxing and modularization. The language lends itself well to design principles beyond those discussed within this report.

Recommendation

Snakemake is the best build automation language choice for individuals and small groups of collaborators. Design patterns and software recommendations can be used to greatly enhance the reproducibility, transparency, and reusability of a shared code-base.

Bibliography

Alexander, C. (1977). *A pattern language: towns, buildings, construction*.

Broad Institute. (2017). *Picard.* Retrieved from github.io: http://broadinstitute.github.io/picard/

Burford, D. (2014, 08 14). *Reasons for using design patterns*. Retrieved from CodeProject.com: https://www.codeproject.com/Tips/808058/Reasons-for-using-design-patterns

Chan, F. C., Hung, S., & Chong, L. (2016). *CLC_Pipelines*. Retrieved from GitHub: https://github.com/LCR-BCCRC/clc_pipelines

Cingolani, P., Patel, V. M., Coon, M., Nguyen, T., Land, S. J., Ruden, D. M., & Lu, X. (2012). Using Drosophila melanogaster as a model for genotoxic chemical mutational studies with a new program, SnpSift. *Frontiers in genetics, 3*.

Cingolani, P., Platts, A., Wang, L. L., Coon , M., Nguyen, T., Wang , L., . . . Ruden, D. M. (2012). A program for annotating and predicting the effects of single nucleotide polymorphisms, SnpEff: SNPs in the genome of Drosophila melanogaster strain w1118; iso-2; iso-3. *6*(2), 80-92. Retrieved from Fly: snpeff.sourceforge.net

Cingolani, P., Sladek, R., & Blanchette, M. (2014). BigDataScript: a scripting language for data pipelines. *Bioinformatics, 31*(1), 10-16.

Conifer Systems LLC. (2010). *What's Wrong With GNU make?* Retrieved from Conifer Systems: http://www.conifersystems.com/whitepapers/gnu-make/

Continuum Analytics, Inc. (2017). *Conda Documentation.* Retrieved from conda.io: https://conda.io/docs/index.html

Di Tommaso, P., Chatzou, M., Baraja, P. P., & Notredame, C. (2014). *A novel tool for highly scalable computational pipelines.* Figshare.

Dobin, A., Davis, C. A., Schlesinger, F., Drenkow, J., Zaleski, C., Jha, S., . . . Gingeras, T. R. (2013). STAR: ultrafast universal RNA-seq aligner. *Bioinformatics, 29*(1), 15-21.

Donaldson, T. (2017, 01 01). *Python as a First Programming Language for Everyone*. Retrieved from www.cs.ubc.ca: https://www.cs.ubc.ca/wccce/Program03/papers/Toby.html

Duell, M., Goodsen, J., & Rising, L. (1997). Non-software examples of software design patterns. (Addendum). *1997 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (pp. 120-124). ACM.

Fangohr, H. (2004). A comparison of C, MATLAB, and Python as teaching languages in engineering. (2004): . *Computational Science - ICCS 2004* (pp. 1210-1217.). Berlin: Springer.

Free Software Foundation. (2016, 05 22). *GNU Make Manual*. Retrieved from GNU: https://www.gnu.org/software/make/manual/

Gamma, E. (1995). Design patterns: elements of reusable object-oriented software. *Pearson Education India.*

Gansner, E. R., & North, S. C. (2000). An open graph visualization system and its applications to software engineering. *Software - Practice and Experience, 30*(11), 1203-1233.

Guo, P. (2014, 7 7). *Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities*. Retrieved from https://cacm.acm.org: https://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities/fulltext

Hume, A. (1987). Mk: a successor to make. *AT and T Bell Laboratories. Computing Science*. Retrieved from Hume, A. (1987). AT and T Bell Laboratories. Computing Science. Chicago

Kanwal, S., Khan, F. Z., Lonie, A., & Sinnott, R. (2017). Investigating reproducibility and tracking provenance–A genomic workflow case study. *BMC bioinformatics, 18*(1), 337.

Koboldt, D. C., Zhang, Q., Larson, D. E., Shen, D., ..., & Wilson, R. K. (2012). VarScan 2: somatic mutation and copy number alteration discovery in cancer by exome sequencing. *Genome research, 22*(3), 568-576.

Köster , J., & Rahmann, S. (2012). Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics, 28*(19), 2520-2522.

Koster, J. (2014, 02 11). *Taming Snakemake*. Retrieved from de.slideshare.net: https://de.slideshare.net/jermdemo/taming-snakemake

Larman, C. (2012). *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interative Development.* India: Pearson Education.

Leipzig, J. (2017). A review of bioinformatic pipeline frameworks. *Briefings in bioinformatics, 18*(3), 530-536.

Li, H., & Durbin, R. (2010). Fast and accurate long-read alignment with Burrows–Wheeler transform. . *Bioinformatics, 26*(5), 589-595.

Li, H., Handsaker, B., Wysoker, A., Fennell, T. R., Fennell, T., Ruan, J., . . . Durbin, R. (2009). The sequence alignment/map format and SAMtools. *Bioinformatics, 25*(16), 2078-2079.

Miller, P. (1998). Recursive make considered harmful. *AUUGN Journal of AUUG Inc, 19*(1), 14-25.

Parker , D. S., Gorlick, M. M., & Lee, C. J. (2003). Evolving from Bioinformatics in-the-Small to Bioinformatics in-the-Large. . *OMICS A Journal of Integrative Biology, 7*(1), 37-48.

Python Software Foundation. (2017). *Python 3.6.2*. Retrieved from Python: https://www.python.org/downloads/release/python-362/

Schwab, M., Karrenbach, M., & Claerbout, J. (2000). Making scientific computations reproducible. *Computing in Science and Engineering, 2*(6), 61-67.

Stallman, M., Roland, M., & Paul, D. (2014). GNU make manual. *Free Software Foundation 3* .

Stallman, R. M., & McGrath, R. (1991). *GNU Make-A Program for Directing Recompilation.* Retrieved from GNU: http://www.gnu.org/software/make/

Sunyé, G., Le Guennec, A., & Jézéquel, J. M. (2012). *Design patterns application in UML. In European Conference on Object-Oriented Programming.* Berlin: Springer.

TIOBE software BV. (2017, 07). *TIOBE Index for July 2017.* Retrieved from TIOBE: https://www.tiobe.com/tiobe-index/

Tutorialspoint.com. (2017). *Python Object Oriented*. Retrieved from Tutorials Point: https://www.tutorialspoint.com/python/python_classes_objects.htm

Wolfgang, P. (1994). *Design patterns for object-oriented software development.* Reading, Massachusetts: Addison-Wesley Publishing Co.