Replacing the Brooks's Law Filter Using Various Abstractions in Java

Danny Mickens
North Carolina State University
dsmicken@ncsu.edu

Ross Eby
North Carolina State University
rneby@ncsu.edu

Michael McMillan
North Carolina State University
mmcmill@ncsu.edu

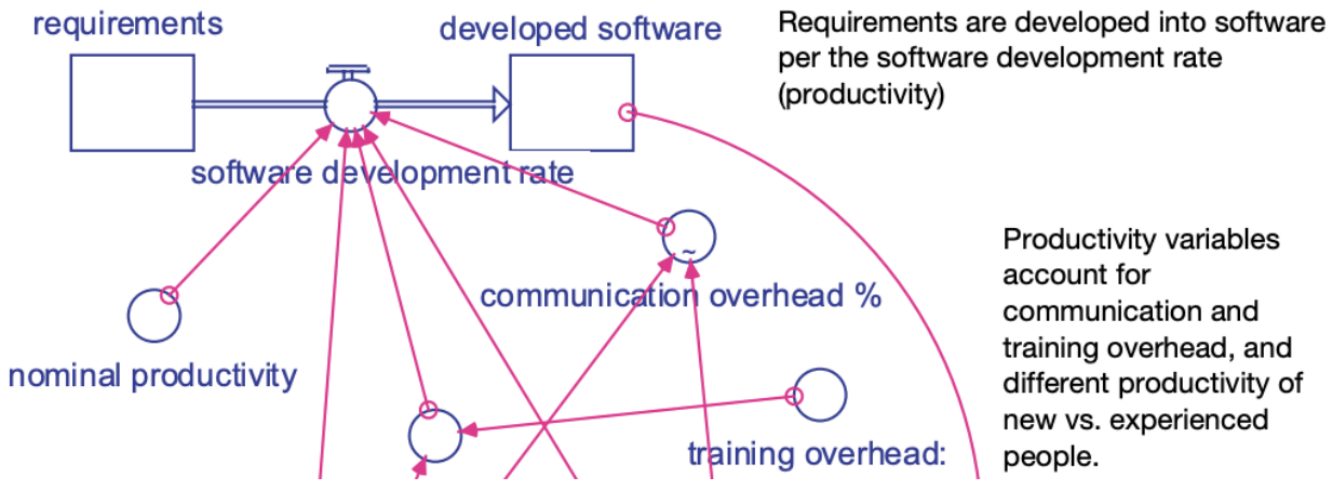

Figure 1: Part of a visual representation of a compartmental model for Brooks's Law

ABSTRACT

This document will explore our choices and their logicality regarding the programming abstractions we have chosen to consider for a Brooks's Law implementation. In addition, we will describe the difficulties we encountered along the way as the details of certain abstractions introduced new problems.

KEYWORDS

programming, abstractions, java, brooks's law

ACM Reference Format:
Danny Mickens, Ross Eby, and Michael McMillan. 2019. Replacing the Brooks's Law Filter Using Various Abstractions in Java. In *Proceedings of ACM Conference (CSC 417).* ACM, New York, NY, USA, 5 pages.

CSC 417, Spring 2019, Raleigh, NC, USA

ACM ISBN N/A…$0.00

1 INTRODUCTION

In project 2A for Theory of Programming Languages we are exploring the use of various programming abstractions while replacing one filter of a pipeline. The filter our group has chosen to replace is the Brooks2 Filter. The Brooks2 filter is a model for Brooks's Law which states that adding programmers to a late project may in fact make it later. Brooks's law is explained by the additional training and communication overhead that gets added when new programmer's are being integrated into the existing team. Brooks's Law is easily modeled by a compartmental model connecting a software development flow chain with a personnel flow chain. Our group attempted to recreate a python implemented compartmental model of Brooks's Law using Java and additional abstractions.

2 ABSTRACTIONS OVERVIEW

Prior to implementation, we initially made a list of ten different abstractions that could prove useful in our implementation. These abstractions include: Inheritance, Delegation, Polymorphism, Pipeline (Pipe and Filter), State Machines,

Compartmental Models, Go Forth, Tantrum, Quarantine, and Map Reduction. These abstractions, or patterns, are simply different strategies to go about implementing a design that serve different purposes and provide different strengths and weaknesses. The details of these abstractions relevant to our implementation will be described in the subsections following.

2.1 Inheritance

Inheritance is a very common object-oriented pattern with a variety of uses. It creates an easily understood hierarchy structure that can be clearly visually understood and designed through a UML class diagram. The main principle behind inheritance is that classes can be structured as super classes and subclasses where subclasses acquire the properties of their super classes. In our code this is exemplified in the Model and Brooks2 classes where Brooks2 extends the model class. This can also be seen in the Stock, Aux, Percent, and Flow subclasses of Thing.

In general, a significant issue with inheritance is due to its nature as a form of "white-box" reuse. "White-box" reuse means that the details of a parent class are visible to the child class. The issue here is that it breaks encapsulation due to the exposure of a child class to its parent's implementation details. The child class becomes limited by the concrete representation of its parent, which is what occurs when a logical inheritance hierarchy is "broken". For example, say we have a parent class called Robot where a Robot is a subclass of Vehicle since a Robot does use wheels to move. Down the line, after creating several children a type of Robot is proposed where they want it to be bipedal. This will not be allowed due to the Robot class defining movement by wheels. Any change to the Robot parent now breaks the other child classes in the hierarchy.

Not all inheritance falls to encapsulation issues, as abstract classes used as a parent in inheritance reduces the dependencies on the parent. The abstract class keeps information about the implementation away from the parent and the children all follow the basic structure of the parent. When used, a client does not need to know the object type to use the same interface provided by the abstract class.

2.2 Delegation

Delegation is another object-oriented specific design pattern which achieves similar goals to that of inheritance through a different implementation. Delegation occurs when an object handles requests by delegating the request to an internal object, called the delegate, that handles the request.

Example: A class called WheeledRobot could have a delegate called WheeledMovementComponent where it delegates all of the movement function requests by calling WheeledMovementComponent's functions. WheeledRobot could be asked to accelerate, then calls accelerate() in WheeledMovementComponent.

Delegation allows for code reuse option that avoids some of the encapsulation issues of concrete inheritance. Delegation allows for changing run-time behavior by changing the instance in the object, so long as they are the same type. Certain "Gang of Four" design patterns, like the Strategy pattern, make use of delegation in this manner.

Delegation does complicate the code more and make it harder to understand. An issue with delegation is that it can easily over-complicate the code if chosen wrongly and over-applied.

2.3 Polymorphism

Polymorphism is the ability to substitute identical interfaces at run time through dynamic binding. It is a cornerstone of object-oriented programming, as it reduces dependencies and simplifies interaction with objects by sharing interfaces. With several objects sharing interfaces, they can be treated with greater uniformity, which can be very powerful in simplifying code.

Example: Given parent class Thing and 3 subclasses named Stock, Flow, and Aux, a client can assume that all subclasses of Thing share an interface with Thing and can treat them all in the same manner. The subclasses can use Function Overloading to provide separate implementations for functions of the same name, which is a form of Ad Hoc Polymorphism.

2.4 Pipeline (Pipe and Filter)

A pipeline is a design pattern where the system is a chain of processing elements called Filters. Each filter is intended to handle one thing and only that thing, where more complex functions are handled by chaining filters together through Pipes. The input and output interface is uniform to facilitate swapable filters.

The pipe and filter design keeps each implementation simple and separate which allows for prototyping, experimentation, and breaking down larger projects. The design is not suitable for highly interactive environments like GUIs where the user is hopping around.

A popular example of the Pipeline design is the UNIX Shell Pipe and Filter system, which uses the stdin and stdout formats for its input and output. The system has a great number of powerful filters that can be swapped with ease, and chained together for more specific functions.

Another example of a system the relies heavily on the pipe and filter abstraction is Microsoft Windows' PowerShell. It

differs from a UNIX Shell in that it takes a more object-oriented approach where it considers computers, users, almost any data type it supports as an object that can be filtered through modules or functions.

2.5 State Machines

A mathematical model of computation that allows for easy to understand logic, while separating the logic out of the program. It consists of a number of finite states and transitions between states in the machine. Transitions have a "from" state, a "to" state, and a "guard" state. The guard state is the requirement the test for the transition from the current state to the next state. States can be simple or complex, from holding booleans to holding classes.

State machines are used for discrete variables, so are not suitable for continuous variables. They are not good for taking care of very large systems on their own, needing to break down the system into smaller sections and implement state machines in smaller portions in order to be useful.

2.6 Compartmental Model

A mathematical model used to describe how materials are moved through a system. It helps keep expression of logic simple and separate it from the program. It is composed of:

- Previous payload
- Next payload
- Stock: variable measure at a time
- Flow: the speed
- Aux: auxiliary variables that affect Flows
- Model: holds the Stocks, Flows, and Auxs in a container and generates the next payload from the previous one

It has a similar role as the State Machine, but is used for continuous variables. It has similar weaknesses as the State Machine, such as becoming overly complex if it tries to model too large a system. Any larger systems should be broken down and Compartmental Models made for each part. A strength of Compartmental Models is the ease of writing them. One can be made by reading a flow diagram and using the general form: payload1.val = payload0.val + dt(sum of all inputs - sum off all outputs).

2.7 Go Forth (Stack Machine)

Named after the Forth programming language, which is an imperative stack-based language. In Forth, local variables were often unused and instead intermediate values were kept in a "data" stack and values for operations kept in a function-call stack. You could also define subroutines called words, to the point that you could redefine the number "2" so that it was 7.

Go Forth is about applying those principles to the system you are using. All arithmetic is done on the data stack, there is a function-call stack for storing later operations, and user defined subroutines. Go Forth can be unnecessary in many applications and can be incompatible with the paradigms that a language is intended for. Language limitations does not mean a language can not do Go Forth, but that it could over-complicate things. A programmer has to carefully manage the data stack, and move data from the function-call stack.

2.8 Tantrum

Every single procedure and function performs sanity checks and will "throw a tantrum" by shutting down if arguments are unreasonable. If an error is found, then the error is propagated up the call-chain and possibly logged in some form.

The pattern only comes into issues when overused and starts to lock the user out in unnecessary situations, but otherwise very useful and widely used. Another potentially negative aspect of this is the potential for slowing down software performance. Making use of continuous integration as well as unit tests can often achieve similar or better software reliability with the performance costs absorbed into the test suite and not the final executable.

An good example of this abstraction is how Java handles typing. Whenever an assignment, re-assignment, or cast is being made in Java, the types of the operands are checked and if they are not in alignment with the pre-determined acceptable or compatible types, Java will not allow it by throwing an exception.

2.9 Quarantine

A style that emphasizes the separation of core program functions and the separation of I/O from pure functions. I/O side-effect code is kept in its own encapsulated sections and all I/O code must be called from the main program. All core program functions must be pure functions, meaning that they have no side effects of any kind (including I/O). These functions must give the same output every time for the same input, similar to a general function in mathematics.

This abstraction is very useful for maintaining encapsulation, preventing code from becoming bloated, and can make it easier to track down bugs in logic due to the functions having no side effects. However, restricting one's self to only writing pure functions can produce non-intuitive solutions to programming problems that may be more difficult to maintain or for other programmers to understand by reading them.

2.10 Map Reduction

Input data is divided up into sections, then a "map" function assigns worker functions to each section of data. A "reduce" function recombines the data from the worker functions into cohesive output. The abstraction is very helpful when

parallelism needs to be exploited, but is harder to debug due to parallelism.

3 EPILOGUE OVERVIEW

Out of our initial list of 10 abstractions as seen in section two we narrowed our abstractions down to three. Some abstractions, while initially seeming useful, proved to be otherwise. Similarly, some abstractions seemed not very useful in this scenario, but after implementing the code it was apparent that it could have been utilized well. Specifically, the Tantrum Pattern seemed like one that would be overkill for checking how reasonable the program arguments were at different stages. After much of the coding had been accomplished it was apparent that the tantrum pattern would have proven very useful in tandem with this compartmental model since there are so many computations using the compartmental model data going on, and the model already had a function in use for checking bounds so the pattern would have fit right in. Nevertheless we ultimately decided on three abstractions to test. Our primary three abstractions that were used were Compartmental Model, Inheritance and Quarantine.

3.1 Compartmental Model

Brooks2 is a Compartmental Model, so it only makes sense to use the abstraction when creating a filter for it. The Compartmental Model was difficult to design for, which ran contrary to our expectations. A lot of our problems came from understanding the python code with python's dynamic typing. It was also quite frustrating working with variables that were not meaningfully named. Trying to keep track of variables with only a single letter, or two or three if we were lucky, was a bit of a headache. We had great difficulty validating our output and figuring out what numbers were suitable to contain in the model. Our difficulty here was partially due to the inconsistencies we found in the write-up for Brooks2. We ended up spinning our wheels often and for long periods of time while attempting to match the input and output with the existing input and output records.

Throughout the implementation however, the compartmental model proved to be a strong and useful abstraction. For Brook's Law especially, it was the right choice. After working with a compartmental model we can recommend that Compartmental Models should be used whenever you have a large amount of dynamic data that is dependant on other related dynamic data. I'll give some examples to explain what this means. When you are manipulating a variable using constants or static variables that don't ever change, that can easily be done without a compartmental model. Depending on the situation, it may just be a one time operation, or a do-while loop if the iterations aren't known, or a for-loop if they are. But regardless, it is not a complex set of operations when you are performing them with consistent data. Compartmental models become useful when the operations that are being performed use dynamic variables. In these cases the operands are being updated throughout the iterative process while moving towards a final state. This leads me to another flag for when compartmental models are useful. Like state machines, compartmental models progress a certain state, or set of data, throughout the model. When you have data that is closely related and the calculations are dependant upon each other, it might be indicative of a time when a compartmental model would be useful.

3.2 Inheritance

Inheritance while making Brooks2 helped create two different hierarchies. One of these was the Thing superclass and its four subclasses: Auxiliary, Stock, Flow, and Percent. The Thing classes describe the different variables that the Brooks2 Model will be using to store the current state. It's subclasses Auxiliary, Stock, Flow, and Percent give more information about the type of Thing that is being stored. Using this inheritance abstraction proved useful in multiple ways. First, the ability to extend the Thing superclass to our Auxiliary, Stock, Flow, and Percent classes saved both time and memory as there is significantly less "boilerplate" code to write when the subsclasses inherit the methods of the superclass. In addition, Java being an object-oriented language allowed us to make use of data structures that can be declared to store a superclass (such as HashMap<Thing>) and be used in conjunction with all types that extend that superclass.

One major disadvantage of the inheritance abstraction is that it introduces tight coupling between the superclasses and the subclasses, thus creating a potentially-avoidable dependency. That being said, if attention is paid to only utilizing inheritance in strict "is-a" class relationships, the cost of this coupling is minimized.

3.3 Quarantine

Quarantine while implementing Brooks2 was so-so in how helpful it was. In general I do like to try and separate distinct parts of the program into their own functions like input/processing/output and any other functions that arise so quarantine would fit into that well. In Brooks2 the input was a dictionary-esque text file with different variables names or keys and values for them which we need for the compartmental model processing. In order to follow Quarantine's guidelines we separated the input into its own method that gets called from main on startup. the input method reads in from stdin or wherever stdin is directed, and puts the dictionary mappings into a HashMap and returns it. No other processes involving the computation of the output are done

until the input method returns and the HashMap can be used, so there are no side effects from the input function. Throughout the compartmental model iterations, a Stringbuilder object is being built and an ArrayList of Strings is being populated with the output that is expected. The ArrayList is returned after iterating and an output method is called with the ArrayList. The output method prints the header and all the information that the output is expected. This output method being separate ensures the otuput is called from main and there are no side effects, both which follow Quarantine coding guidelines.

Quarantine seems like it would be useful often but not always. It was useful for handling the input in Brooks2 because the input needed to be formatted and stored differently than it was arriving. So I believe that whenever you know the format of the input or you know you will need to manipulate the input before using it, Quarantine is useful. However sometimes the input may be small amounts of data or something as simple as a boolean or a single number. Not much is being done with the input, but you just need to know its value. In these cases, Quarantine just becomes extra work and provides no value to the program structure. On the output side of things, Quarantine was not ideal for Brooks2. It added extra steps that were not needed to solve the problem. Initially we implemented the output without considering Quarantine and had to change it. This is because handling the output was already an intuitive process that didn't occur all at one time like Quarantine would expect to happen from the main program. I would suggest that Quarantine be used in a situation similar to the input reasoning but for output: use it when you have to manipulate the data you have to match a known output format. Our implementation generated strings and arraylists of strings to hold on to the output until it could be returned to main and sent to an output function. Since quarantine restricts output calls to be from main, this restricts output to be generated all at once or you must have some kind of looping/iterative process happening in your main program. If this doesn't fit well with your program structure, I would advise against it and go with an intuitive output process.

4 EXPECTED GRADE

For this assignment the grade considers a few things: the code implementation, the paper, the filter, the abstractions used, and the language. First off, the paper. We followed all guidelines and formatting restrictions for the paper. All content should be there and comprehensive for the allotted space. I think the paper is worth full credit. The language we used was java which is a one star language so no bonus points are earned there. The abstractions we used vary, there is a one star, two star, and three star abstraction. Assuming those bonus points stack that would be worth 3 bonus points, if only the highest abstraction is taken into consideration then that would be 2 bonus points, and if it is an average of the three abstractions then it would be 1 bonus point. The filter was Brooks2 which was a 3 star filter and worth 2 bonus points. The rest is tricky. Our group has a very strong collective agreement that the Brooks2 assignment description and code give us an unfair opportunity to implement the code correctly. The description on the Brooks2 page and the source code do not match, and neither of them work correctly individually. On the Brooks2 page, there is data that is missing and in one calculation, softwareDevelopmentRate, variables are used that don't exist. They are never described or created previously and then out of the blue they are being used. In the source code, non documented constants are used in place of those, so that's what we went with. However, in the source code, constants are used in places where the input file/stdin already provides values for, so then they are never used. We understand the process that our code must follow: read in input, turn it into a model, create an initial payload holding state meta-data, and run the payload through time/dt iterations to get a final state that is bounds checked and restrained. We attempt to do this but without knowing what the correct values and variables are that are being used, we cant be sure we are producing the right output. We were told we would lose 4 marks for being late, but we were only late because we were trying to solve a problem that doesn't seem to be able to be solved. At the time this was initially due we had a wiki page of our 10 abstractions, we had chosen 3 to implement, we had started the paper, and we had attempted the code. We could not get the code to work and thus couldn't finish writing the paper. I believe we deserve to at least not lose the full 4 points, if not lose none at all.