Replacing the Monte Carlo Filter Using Various Programming Abstractions in C++

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

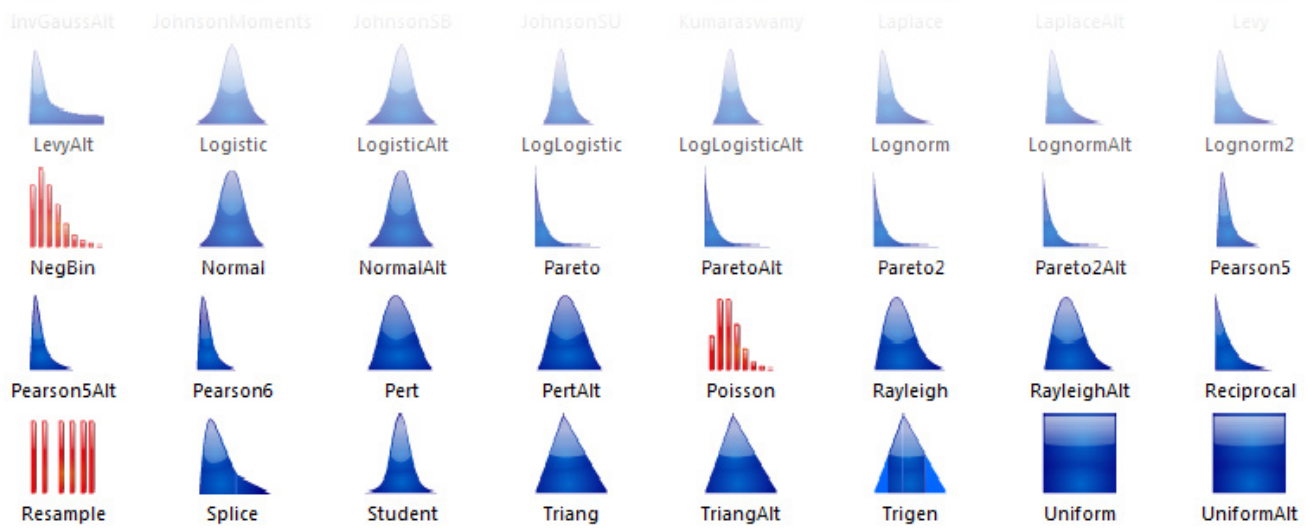

Figure 1: Common probability distributions

ABSTRACT

This document will explore our choices regarding the abstractions we used for our Monte Carlo implementation, as well as other abstraction considerations we have made along with those abstractions' pros and cons. Additionally, we will discuss the unique challenges presented by implementing this behavior within the chosen abstractions' parameters.

KEYWORDS

programming, abstractions, c++, monte carlo

CSC 417, Spring 2019, Raleigh, NC, USA

ACM ISBN N/A…$0.00

ACM Reference Format:
Danny Mickens, Ross Eby, and Michael McMillan. 2019. Replacing the Monte Carlo Filter Using Various Programming Abstractions in C++. In *Proceedings of ACM Conference (CSC 417).* ACM, New York, NY, USA, 5 pages.

1 INTRODUCTION

In part 2B for our Theory of Programming Languages project, we are considering various programming abstractions for replacement of the Monte Carlo filter in the provided pipeline. The Monte Carlo filter provides the functionality of a Monte Carlo Simulation which is a technique useful for generating a set of random samples over which one may run an algorithm. The method by which a Monte Carlo Simulation is run can vary, however they all follow the abstract pattern:

(1) Define the input domain
(2) Using a probability distribution, randomly generate a set of possible inputs
(3) Using those inputs, perform some deterministic computation
(4) Process/interpret the results of the computation

Our group has recreated the given Python implementation of Monte Carlo using C++ with various programming abstractions that will be discussed in this document.

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, Macros, 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. [1]

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 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 Graphical User Interfaces where the user is hopping around from function to function, because pipe and filter is meant to handle data in one direction only.

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 Macros

A "macro", which is shorthand for "macro-instruction", defines how some input is mapped to a replacement output, which is known as macro expansion. For example, we could have a macro called Square(X) that takes in a parameter X, and expands to evaluate X squared. Macros can be used to shorten repeated blocks of code by using the macro in its place. Through this, Macros allow for code reuse or domain-specific languages.

The main drawbacks of using macros are debugging them and the mental load on the programmer. Macros are not like functions and are a separate skill, so beginners tend to write buggy macros. For example, a common issue is variable name capture, where the macros variable names conflict with the local variables names in the section of code this is called in. The solution here is to generate unique variable names for the macros, but it is still a common error. Macros also put an extra mental load on any new programmers, because it is a new programming concept that the programmer needs to understand. In general, the rule for macros is to be very conservative in creating them to avoid spaghetti code and debugging issues.

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 proved very useful in tandem with the Quarantine abstraction since the role of the MonteCarlo filter is to generate a data set for other pipes to use. Ensuring correct input and output of the MonteCarlo filter is necessary, and at no point can it be erroneous. We ultimately decided on three abstractions to test. Our primary three abstractions that were used were Quarantine, Tantrum, and Macros.

3.1 Quarantine

Quarantine while implementing MonteCarlo was fairly useful. In general, it is good practice to 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 MonteCarlo, the input were optional arguments to run the filter with. One option was to set the number of repeats while generating the data, used by a "-n [int]" when running the filter. The other option was to set the random number seed by using a "-s [int]" when running the filter.

In order to follow Quarantine's guidelines we separated the input into its own method that gets called from the main method on startup. The input method reads in from stdin or wherever stdin is directed, and puts the command line arguments into the appropriate variables if they are correct. No other processes involving the computation of the output are done until the input method returns and the seed and number of repeats can be used, so there are no side effects from the input function. Throughout the MonteCarlo iterations in the main method, an unordered map object is being built and then sent into an output function. The output method prints all the information that MonteCarlo is required to generate. This output method being separate ensures the output is called from main and there are no side effects, both which follow Quarantine coding guidelines.

Quarantine was very useful for MonteCarlo. It was useful for handling the input in MonteCarlo because the input needed to be formatted and stored differently than it was arriving, and input arguments were directly tied to the numbers used to generate data. 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. In this case, we knew the format of the optional arguments that were needed, as well as bounds for the possible values of each argument. For example, we could not allow the number of iterations to be negative. Quarantine assists in neatly separating this code to handle error checking. One thing that we have discovered while working with Quarantine is that maintaining your program's dynamic data in a way that it is easily accessible and manageable gives you a very easy way to expand the functionality of your program. For example in our program we are keeping everything together in an unordered map whose keys we already know. We can access any part of the data at any time using these keys and the map, so if there are any new processes we want to run this data through or if there is a new way we want to decide to store this data, it is incredibly easy to implement because the quarantine-valid input method provides us with easily accessible source of data that can be passed around, duplicated, or updated at any time.

On the output side of things, Quarantine was fine for MonteCarlo. Handling the output was already an intuitive process that occurred once an iteration from the main method, and all at one time after calculations were complete, just like Quarantine demands. 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 for the output function printed formatted strings to the output once per iteration, as long as the verbose setting was on. 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. In our case, we had the latter restriction. Although, if this doesn't fit well with your program structure, we would advise against it and go with a more intuitive output process.

3.2 Tantrum

Tantrum was used in conjunction with Quarantine to ensure correct input and output. In MonteCarlo the major sources of error come from the input arguments from the user. With the two arguments for the number of iterations and the random number seed, if the numbers go unchecked they will

break the filter. For example, if the command line argument "-n 0" is used, there will be no iterations and the data will never be created. Tantrum allows the filter to reject input entirely if it is unsuitable to run, which is ideal for this type of application. It was easy to implement, but the sections for it within the input and output functions are bloated due to all of the conditional checks. Luckily, some of the macros created allowed for extensive code reuse in these sections to reduce bloat.

Another source of data bounds checking occurs in the doubleRand() function. In an attempt to make this program more platform independent as well as more reusable under different C++ compilers, gnu's g++ being the primary compiler we used for this program, the randomly generated value from 0 to 1 is double checked before returning to make sure it satisfies the bounds generated by the current environment. RAND MAX is guaranteed to be at least 32,000 and some change but can be much much higher, meaning that in certain environments and under very unlikely conditions, the precision of our random double between 0 and 1 may be outside the scope of the precision guaranteed by a double. For this reason the bounds are checked and the function is recursively called until a suitable number can be returned.

3.3 Macros

Macros were used in our implementation of MonteCarlo mostly for the sake of code reuse. In the input and output functions created by the Quarantine abstraction, the Tantrum style error checking became a bit unwieldy and long. To reduce the code length and duplicated code, we created many macros.

Many of the error messages for Tantrum were the same, so a PrintErrorReport macro was created that prints out the error message. The macros we created dealing with output were helpful in allowing code reuse while maintaining the Quarantine abstraction. An alternative to macros in this way could be private functions, but that would violate the Quarantine principle that all input and output code must be called within the main method. The fact that macros expand into the blocks of code, instead of being a separate function call, avoids Quarantine violation.

In the input methods, there were a multitude of initializing statements for the variables handling the number of iterations and the random number seed. We created macros INITSEED and INITNUMREPEAT to handle both statements and their duplication through the code.

Although the macros were helpful, there were implementation issues that were described in the abstractions section for macros, namely debugging and limitations of macros in C++. There was an issue where we wanted a macro to do the conditionals for error checking within the input and output functions, but C++ was not happy. After some struggles and investigation, we realized that C++ macros does not support if-else conditionals in its macros. Macros in C++ do not have the same kind of access to the full functionality of the language that a language like LISP has, so there are limitations on how powerful the abstraction can be within languages. If C++ did have full access to its features in the macros, then we could have reused whole if-else blocks and condensed the input and output functions down greatly.

C++ has lackluster macros, whereas LISP, Julia, Rust, and Clojure all have macros that allow for full functionality of the language. The macros abstraction is far more useful if the programming language has a robust macro language.

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 C++. We did not get a timely reply from Professor Menzies to determine whether we could use the language or not, so we went ahead with the implementation with the rationale that C++ is similar to other one star languages. The abstractions we used were all three star abstractions. Assuming those bonus points stack that would be worth 6 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 2 bonus points. The filter was MonteCarlo which was a 1 star filter and worth no bonus points. The maximum possible total points would be 16 points.

An abstraction that we could see being a point of contention is the use of the macros abstraction. Although we did make great use of macros, the project write-up did technically state that the macros must implement a domain specific language and be in a pre-approved language. We were unable to acquire permission for C++ as implementation language, much less for implementing a domain specific language. Depending on Professor Menzies' discretion, we expect anywhere between 14 and 16 points on this project.

REFERENCES

[1] Mickens, D., Eby, R., and McMillan, M. Replacing the brooks's law filter using various abstractions in java.