Project 2

S. A. Bencherif*
J. D. Benckert*
J. Rodriguez*
sbenche2@ncsu.edu
jdbencke@ncsu.edu
jrodrig6@ncsu.edu
North Carolina State University
Raleigh, NC

ABSTRACT

We have spent the month of March exploring various avenues of software development. In this time we have uncovered a sea of philosophies and schools of thought–each sporting their own unique strengths and weaknesses. Thus the task of narrowing a selection of programmatic paradigms is rendered as daunting and exciting as purchasing a new car. Consider the following document a vehicle history report for each of our top ten programming abstractions. In this report we will be covering some broad "pros and cons" as well as an applied analysis of the structures in respect to our particular use cases. Along the way we will provide a top-level explanation of the concepts in a allegorical and literal sense. That is, we will attempt to encompass the ideas involved in an abstract sense that could be suitable to even a layman and follow that with concrete examples that involve actual code (or pseudocode).

CCS CONCEPTS

• **Software and its engineering → Language features**; *Polymorphism*; *Inheritance*; *Modules / packages*; *Patterns*; *Macro languages*; *Software prototyping*; Control structures; • **Theory of computation →** *Lambda calculus*; *MapReduce algorithms*; • **Hardware →** *Finite state machines.*

KEYWORDS

domain specific languages, programming paradigm, design patterns, software engineering

ACM Reference Format:
S. A. Bencherif, J. D. Benckert, and J. Rodriguez. 2019. Project 2. In *PLM19: Project 2, March 01–31, 2018, Raleigh, NC*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*All authors contributed equally to this research.

PLM19, June 03–05, 2018, Raleigh, NC

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$0.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 STATE MACHINE

A State Machine is a theoretical representation of computer hardware, initially conceived in 1969 by Knuth, Morris, and Pratt. It is an essential mechanism that brought forth a genesis for formalized proofs regard theoretical computational operations. The State Machine has value beyond that of an abstract mathematical object. Finite State Machines, and typically quite smaller ones, are used as a programming strategy. This programmatic system can be thought of as a conceptual machine that stores a list of states, transitions between states, and a supporting programmatic layer making it far easier to codify certain algorithmic ideas. State Machine's are especially useful when "memory" is a relevant aspect of an algorithm. *Memory* in this case is referring to the algorithmic property of intelligently using prior events when dealing with objects in a sequence. One example might be to efficiently check for a hard coded sub-string in a character sequence; this could be achieved by using a state for each matching character in the string. That specific strategy is essential in hardware programming, such as for garage door openers. Another case might be to manage a graphical user interface, in which case transitions would represent user-events and states could represent display windows. State Machines are not suitable, however, when there are more than, say, a dozen states to manage–at that point the abstraction would be likely tend the whole operation towards "glorious and terrifying hack" as opposed of simplifying things. In our specific use, we made use of a very basic state machine. We used an iterator, which matches the profile of a state machine with n-states for n = number of elements in the array. In our programming we used a *switch case* control structure to delegate our transition routes. Though this served us well, our use only takes advantage of a slice of the benefits that we can expect from State Machines. Below we provide an example of a simple state machine implemented in C.

```
int entry_state(void);
int exit_state(void);

int (* state)(void)[] = { entry_state, exit_state};
enum state_codes {entry, end};

enum ret_coded{ok, fail};
struct transition {
    enum state_coded src_state;
    enum ret_codes ret_code;
    enum stat_codes dst_state;
```

```
};

struct transition state_transitions[] = {
{entry, ok, entry}
{entry, fail, end}};

#define EXIT_STATE end
#define ENTRY_STATE entry

int main(int argc, char *argv[]) {
    enum state_codes cur_state = ENTRY_STATE;
    enum ret_codes rc;
    Int (* stat_fun)(void)
    while (1) {
        state_fun = state[cur_state];
        rc = stat_fun();
        if(EXIT_STATE == cur_state)
            break;
        cur_state = lookup_transitions(cur_state, rc);
    }
    return EXIT_SUCCESS;
}
```

2 POLYMORPHISM

Let us imagine you are about to head into battle. You need a horse to go into battle. It must be a strong, fast, and courageous animal. You encounter camels. You are warrior, not a zoologist, so to you they look the same to you. Thus you ride to battle on camel-back. That is Polymorphism. Both creatures can be used in the same way, though they interface with their environments in significantly differing ways. The camel can retain water and traverse sand, whereas the horse is much more capable in colder or grassy environments. In programming, Polymorphism means that objects are permitted to have some differences so long as they may be used in more or less the same way. For example a file stream object would need operations for reading and writing characters. These operations would be exposed to the client program. Whether or not the steam is a file, network, or memory location would not need to be shared to the underlying subsystem or even higher level programming. Polymorphism is a critical part of Object Orientated Programming, which many programmers consider to be one of the easiest design patterns to grasp. Some significant advantages to Polymorphic programming is how easy it becomes to create and prototype new features as well as maintain large projects. A equally significant disadvantage is how easy it is to get carried away with Object Orientation, resulting in obtuse, hard to manage, and inefficient code. Unfortunately, partially due to sub cultural affinities it is common to see poorly managed Object Orientated code. For this reason it is important to be mindful when using this approach. This phenomena has been far better, and more appropriately, discussed by others, such as contemporary programming scholar Brian Will in his famous lecture "Why Object Orientation is Bad". In our project we put significant consideration into Polymorphism and through our deliberations determined it would not be appropriate for our project. An important part of making use of programmatic structures and strategies is to know what not to use. In our example, we decided that the scope of our project was narrow enough that to make use of Polymorphic properties would, so to speak, over stack our development scheme. We include an explanatory programming demo (aptly in Java code), that shows how two similar objects can interact in the same ecosystem.

```
Class rows {
    public int numNorm(a,b){
        return a*b;
    }
}

Class dom extends rows{
    public int dom(t,a0,c){
        a0 = numNorm(t.nums[c], a0);
        return a0;
    }
}
```

3 INHERITANCE

There are times in programming where it is necessary to create a program made up of various closely related moving pieces. Using classes these pieces may be stateful, and using Inheritance they may rely on each other. Inheritance has the highly relevant feature of helping to avoid the need for repeating functions and data structures. Repetitive code would not only be tedious but also slow a program execution, increase binary file size, and provide undue development challenges. It is also far less beautiful. Luckily, this is where Inheritance can come into play. Inheritance is a part of the object-oriented paradigm. The main purpose of Inheritance is to tether one or more classes to a main super class. This means that the subclass inherits or has access to fields and methods of the parent class. One major advantage is how this makes it easier to reuse code. Inherited classes can have full access to code in the parent class without the need to redefine the code. This can also make it possible for the subclass to make necessary changes to methods or variables in a way that fits their needs without the changing the parent class. However, this also brings out a major disadvantage. With inheritance, the classes are tightly coupled. Even if a subclass only needs one method or variable from the parent class it must include everything from the parent class, making the program go through more and more layers than needed, thus wasting unnecessary processing time. Having the classes use inheritance also makes them highly dependent on each other; if the parent class fails or changes it also affects all classes that inherit from it which can be critical in project maintenance. In our project we used Inheritance for our *Monte Carlo* class. It has served us well in reducing development effort. Our Java example code demonstrates a class *row* that inherits from a super class *num*.

```
class num {

    num(n)
    {
        this.n = n;
    }

    public int inc(){
```

```
        this.n++;
        return this.n;
    }
}

class row extends num {
    public int inc(){
        this.n += 2;
        return this.n;
    }
}

class Main {
    public static void main(string[] args)
    {
        num a = new num(10);
        num b = new row(10);

        System.out.println(a.inc());
        // 11

        System.out.println(b.inc());
        // 12
    }
}
```

4 LAMBDA CALCULUS

Lambda Calculus is formal system of computational logic that combines functionalism with anonymity and often atomic levels of abstraction. In a *purely* lambda calculus language there is no global state and there are no side effects. All information is passed in and out of anonymous functions. The benefits are clear. The formalism allows for highly rigorous development which is invaluable on success-critical missions. Additionally lambda calculus is highly compatible with recursion which is very naturally occurring in the world of theoretical computer science and mathematics. The downside is also obvious. Lambda Calculus is difficult for both humans and computers. There is so much "motion" going on that the symbols may require a new developer lots of backtracking. Likewise the reliance on function calls will result in a dramatic memory hit for any intensive applications. Our sample is in Python. It shows how to compute a factorial and does so without using any Python language features that wouldn't be available in any other Lambda Calculus language. (Except of course the operators and operands, but we encourage the reader to refer to Church with any further questions.)

```
# factorial of 7
(lambda i: (lambda f: f(f)(i))
(lambda f: lambda n: n*f(f)
(n-1) if n>1 else 1))(7)
```

5 MACROS

Macros, or macroinstruction, is a pattern used to specify how a certain input sequence should be mapped to a replacement output sequence. Through this process, a programmer is able to make tasks in the program less repetitive through code that can be reused. One major advantage of macros is that are only expanded and utilized after the compiling time. This helps save in processing time and memory space making macros very efficient in utilizing resources. However, the use of macros is only effective to use for simple tasks in a program. Trying to use macros for complex tasks is very tricky and can usually lead to messy problems for those who donâĂŹt have a lot of experience.

```
Marco doms (t, n,c row1, row1, s)
N = leam.dom.samples
C =t.nam+1
Row2 = another(r1, t.tows)
dump(t.rows)
```

6 PROTOTYPING

Prototyping is a process that involves creating incomplete versions of software programs being developed. Doing prototyping is done to simulate a few aspects of the final product that can be completely different. What makes this key is how it allows a designer to gain feedback from the product and gain some insight on accuracy and estimates on the program. Prototyping is a unique form of testing that has many advantages. With gained feedback of accuracy and estimates, this can reduce the time and costs in creating programs. However, prototypingâĂŹs results may not always reflect the actual quality or accuracy of the program. This leads to the disadvantage of prototyping of leading to insufficient analysis. An improper analysis can also lead to the problems of excessive development time and developers misunderstand the objectives of the program.

```
class rows{
    public int size(n){
     return n;
    }
}

class dom extends rows{
    public boolean testSize(n,m){
        if(size(n) > m)
            return true;
        else
            return false;
    }
}
```

7 MAP-REDUCE

Map reduce is a programming model for processing and generating data sets with a parallel, distributed algorithm on a cluster. The basic structure of a map reduce structure involves a method that performs filtering and sorting on data and a reduce method that performs a summary operation. This model is a specialization of a split apply combine strategy in data analysis. One advantage of using map reduce is it provides a range of data sources making it easier to access and arrange this data for the program. Having a broader range of data sources also provides a cost effective way of handling data freeing memory space and processing time. However,

there are some issues when using map reduce. Map reduce is only effective when dealing with large batches of data. Dealing with small data environments can lead to redundancy and a less effective work environment.

```
public class rowCount {
   public void map(longwritable key, text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOExcepti
        string line = value.tostring();
      stringTokenizer tokenizer = new StringTokenizer(line);
       while(tokenizer.hasMoreTokens()){
           word.set(tokenizer.nextToken());
           output.collect(word, one);
       }
    }
}
```

8 RULE BASED

Rule based is a system used to store and manipulate knowledge to interpret information in a useful way. A basic system is made up of a list of rules, an inference engine, temporary working memory, and a user interface. The main idea behind a rule based system is to derive execution instructions from a starting set of data and rules. A major advantage to this system is this makes it easier to interpret data based on the rule. This can save in processing time making the program more efficient. However, this can also lead to a disadvantage. If there are too many rules this can reduce the efficiency of the interpretation making the program less efficient. This can also lead to problems in terms of prediction quality.

```
rules()
{
    char done = FALSE;
    while (done == FALSE)
    {
        done = TRUE;
        if (count < size)
        {
            scan = TRUE;
            done = FALSE;
        }
        if (count > size)
        {
            scan = FALSE;
        }
    }
}
```

9 DELEGATION

Delegation is the process of evaluation either a property or method of one class in the context of another class. In some ways this is like inheritance but with a few key differences. Instead of having a class coupled with another class it simply forwards some of its methods/variables to another class. One key advantage to this is that a class isnâĂŹt dependent on a parent class making it more flexible. The class also doesnâĂŹt have to utilize all methods and variables of another class only using what it needs; making the program more efficient without the need of going through multiple layers. However, one disadvantage is how it is unable to use passed down methods and variables in different ways that suits its purposes making it unable to use dynamic polymorphism. Another problem is how not all object oriented languages support delegation making it less flexible when dealing with programs of other languages.

```
class row
{
  public int size(n)
  {
    return n;
  }
}

class dom
{
  row r = new row
  {
    public int size(n)
    {
      r.size(n);
    }
  }
}
```

10 PATTERN MATCHING

Pattern matching is a process of checking a given sequence of tokens for the presence of some form of a pattern. What is key in pattern matching is that a sequence of tokens have to be an exact match to count as a pattern. One of the main uses of pattern matching is to create tree patterns that can be used to process data. An advantage to this is it makes things simple and efficient lowering processing time for the program leading to a more efficient program. However, this also leads to a disadvantage. Because the use of pattern matching is more simple this means that it lacks some features available in regular expressions. This causes pattern matching to be less flexible.

```
Public abstract class List<T>{
 Public class Nil : List<T> {  }
 Public class cons : List<T>
 {
   Public readonly T Item1;
   Public readonly List<T> Item2;
   Public Cons(T item1, List<T>, items2){
     this.Item1 = item1;
     this.Item2 = item2;
   }
 }
}
```

EPILOGUE

When working on converting the monte carlo code from lua to java we had to think about the natural structures of the two languages. Thankfully, the two languages had quite a bit in common in terms of structure. The lua language itself is centered on being object oriented. What this means is how a program is treated like an object that other programs can create and update as needed. Some of the basic parts of an object oriented program made up lua like a constructor method, a main method, and various helper methods that help to perform the role of the program. Java features many of these same kind of methods giving us a good step forward when converting the pipeline from lua to java. Despite featuring similar structures their were some challenges we had to face.

One issue came with implementing the constructor method. In lua a programmer could not only define default values to variables in the program, but also attributes to these variables that can help provide needed limitations that can dictate the type of information or range they can take. Sadly, java doesn't have these same features built in to the language. It was important that monte carlo not only have these variables but the attributes as well. When thinking about possible ideas, one possibility was to not only create these variables but variables to represent the attributes to compare to when creating the problem. While the solution could work, it would not be the best to use for the program. Because the program would need to compare the variables to these comparison variables every time a new object was created or an object was changing its variables. This would lead to wasted processing time causing the program to be slower than what was most efficient. Plus, the extra comparison variables would take up memory that could be used in compiling.

We eventually came up with the solution to use a hash map. With its simple nature, we would have the variable stored on the left side while having the attributes on the right side. Having the variables stored like this we could have the variables set with their own value and attributes, yet it would still be easy to access while being efficient with memory to save on compile time. One thing to note was how to represent attributes to add to the hash map. With this in mind, we created a new class that would represent the variable attributes. The class makes it possible for variables to have attributes that java usually lacks.

Another issue we had to deal with was the arguments that would come in for the generator. In lua, it would use an argument parser in order to handle these various arguments. However, java has no access to this parser. To remedy this we created a new argument parser class called arguments in order to handle these arguments.

Designing the arguments class itself proved to be another challenge in itself. Each argument that the pipe receives has its own set of actions that they do that can differ from the others. We would need a way to not only to take in arguments but also do different actions depending on the argument. One possible solution would be having the class doing certain actions based on the argument and having a argument class created for each type of argument. This would be a problem though since having all those classes at once could cause a slow down in processing and compiling time. To fix this issue, we decided to implement a state machine in the class. Having a state machine would allow the program to have just one argument class that can switch on the fly how it performs an action depending on the argument given while saving on memory space for a more efficient processing time.

Max Grade Expectations. For the language we used java which is a one star language for a total of zero bonus marks. For the filter, we went with the monte carlo filter which is a one star language for zero bonus marks. As for the abstractions, we went with trying state machines for one bonus marks, polymorphism for zero bonus marks, and inheritance for zero bonus. All this totaling one bonus marks for abstractions. There is then the initial ten marks for having the filter change working properly. Adding everything up and having all functionality working without issues, the total amount of marks should be equal to eleven marks.