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 multiple 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 your top ten programming abstractions.

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, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

A State Machine is a both a theoretical representation of computer hardware and a useful programming strategy. The structure maintains a machine to store lists of states, transitions to move between states, and states themselves that can make it easier to codify certain algorithmic ideas. State Machine's are especially useful when "memory" is a relevant aspect of an algorithm. One example might be to hyper efficiently checking 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. Another case might be to manage a graphical user interface, in which case transitions would represent events and states would 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 doing the opposite of simplifying things. Below we provide an example of a simple state machine implemented in C.

*All authors contributed equally to this research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLM19, June 03–05, 2018, Raleigh, NC
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$0.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 STATE MACHINE

```
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 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)
```

3 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 are equivalent in utility. Thus you ride to battle on camel-back. That is Polymorphism. Objects are permitted to have some differences and be used in more or less the same way. For example a file stream might 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 be shared. that information is irrelevant to how the client should use the object. Polymorphism is a critical part of Object Orientation, which many programmers consider to be an easy to grasp and essential programming method. A significant advantage is how easy it becomes to create and prototype new features. 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. This has been well discussed by others in the past. Our example, in Java, shows how two similar objects can interact in the same ecosystem.

```
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 INHERITANCE

A feature that allows a class to inherit the attributes and methods from another class. This involves a subclass being the class that inherits from another class or the superclass.

```
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;
    }
}
```

5 MACROS

A program that is used to expand functions and variables in the program. They are used to write programs during runtime in another program.

```
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

A process that creates object by copying a prototypical instance of them. They are used to.

```
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

A model that involves processing and generating big data sets with parallel, distributed algorithm on a cluster.

```
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

A system that can store and manipulate knowledge in order to interpret information. Using the system involves deriving execution instructions from a starting set of data and rules.

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

9 DELEGATION

A process that evaluates either a property or method on an object/class in the context of another object/class. This usually involvolves passing the sending object to the receiving object.

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

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

10 PATTERN MATCHING

A process that involves checking a given sequence of tokens to see if they match a certain pattern. It is important to not that the match usually has to be exact.

```
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

One of the first challenges we faced was understanding the typescript language. The language itself wasnâĂŹt too hard to understand due to past and current experience with it. What proved tricky was converting the lua language encompassing the dom class and the classes associated with it. Because typescript is more of an object oriented based language compared to lua, we had to apply a more rigid structure to our code. This made things tricky when taking into account the more fluid and loose structure of the lua code. We also had to take into account that fact the data storage

was more fixed in size. One piece of advice that is important to keep in mind when changing the language of code to another; try to have the languages similar in structure for an easier transition. Like changing a lisp based language with another lisp based language, or an object oriented language with another object oriented languages..

Because the dom class uses methods and variables from other classes, we had to take into account several abstractions that would make is possible. One of the abstractions we tried was inheritance. Given the fact the object oriented nature of typescript it would seem like inheritance would be the best way to go for passing along methods. Yet, we noticed that the dom class only used a few of the methods and variables it inherited. But, since dom needs to have contact with the classes it inherited the methods at all times it is was a waste of space and a slower processing time. We eventually tried to do delegation. While like inheritance it differs in that it passes a version of the class to the other class. This made it so that we didnâĂŹt always have to have contact with the other class saving data and time. Interesting enough, the lua classes actually used delegation frequently. One tip is that while inheritance and delegation have their similarities one may be better than the other depending on how many functions/variables from one class you want to use in another.

Unexpectedly we found a couple more abstractions came into play. Due to differences in Typescript and Lua we had to write some intermediary code. Namely, the way concurrency works is very different. As Javascript was originally designed for the web browser, and almost always run in a single thread, thread blocking is very much not allowed. This is normally not a problem, however a core functionality of our program revolves around taken standard input. To ameliorate this issue we created a bidirectional Promise queue that is responsible for keeping

```
io.read()
```

calls in sync with the rest of the program.

Max Grade Expectations. For the language we used typescript which is a three star language for a total of two bonus marks. For the filter, we went with the dom 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, lambda calc for two bonus marks, and polymorphism for zero bonus. All this totaling three 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 fifteen marks.