Project 2A

Group G

Josef Dewberry
CSC 417
North Carolina State University
Raleigh, NC, USA
jadewber@ncsu.edu

Thea Wall
CSC 417
North Carolina State University
Raleigh, NC, USA
adwall4@ncsu.edu

Spencer Yoder
CSC 417
North Carolina State University
Raleigh, NC, USA
smyoder@ncsu.edu

ABSTRACTIONS

Here is a list of ten abstractions. The first three are the ones that we focused on in our project.

1.1 State Machines

A model made up of states. The machine can transition from one state to another depending on a given input. A machine is defined by its initial state, a list of its states, and the conditions for each transition.

Example: A traffic light has three states: green, yellow, and red. After a certain period of time the light will switch from green to yellow, yellow to red, and red back to green.

1.2 Inheritance

A system in which objects or classes “inherit” the properties and methods of other objects.

Example: To build a collection of people at a university, I would include professors, grad students, and undergrad students. Undergrad and grad students would inherit the ability to have classes from the professor object, and undergrad students would inherit the ability to get a degree from grad students.

1.3 Polymorphism

The ability to process objects differently based upon their type or class.

Example: I could create several different methods with the same name, such as 3 methods called print(), but they all took different parameters (print(), print(int), print(string)) I could call them without specifying which method. The appropriate method would be called depending on what parameters I put in the method call.

1.4 Delegation

Passing off responsibility to something else to keep objects from becoming too complicated.

Example: I could create an object with an add method, and then that add method really just called an add method from a different object.

1.5 Layers

The ordering of code in a program in “layers”, or a hierarchical system. Generally each “layer” can only interact with the layers above and below it.

Example: Operating systems can be made in layers to keep the code simple. From the base layer to the top it can go hardware -> CPU scheduling -> memory management -> operator console -> input/output -> user programs -> operators.

1.6 Pattern Matching

Looking for and finding sequences of data that form a pattern amongst raw data or tokens. The match must be exact.

Ex: Pattern matching allows for us to check if there are multiple methods that are the same (despite being written different or in different languages).

1.7 Pipes and Filters

A pipe is made up of many filters. Each filter takes some input and gives some output. Each filter has one job. Each filter is linked to another filter, essentially "piping" the input until the end when the final output is given.

Example: Compilers are generally pipes with filters for analysis, parsing, and code generation.

1.8 Iterators

A generalization of a pointer which can run through the items in a list or container. Iterators allow you to interact with each item in the list without having to progressively call and find each item.

Example: A while loop that prints every item in a list could use an iterator. While the iterator has another item it can go to, print the current item and change the iterator to the next item.

1.9 Interpreters

A program that executes code without having to have it compiled into a machine language first.

Example: Self modifying code, which can be accomplished using LISP macros, are more easily done with an interpreter rather than a compiler.

1.10 Macros

Instructions that turn a certain sequence of code into a different sequence of code.

Ex: Using #define to create global variables is a Macro. I could do #define PI 3.14 and the compiler would search for every instance of PI and replace it with 3.14. If I wanted to change PI to 3.142 then I would only need to change one line of code instead of many.

EPILOGUE

The input to the dom program is a table made up of columns. But the columns must be handled differently based on their type. In this way, we used **Polymorphism** to handle different types (Number or Symbol) in different ways. The Column class has abstract methods add() and toString() which are handled differently in the classes NumberColumn and SymbolColumn.

These classes also use **Inheritance** to share the properties of Column. This includes a String field for the column header and an integer field for the size. This allows us to abstract fields away from the concrete classes.

The **State Machine** pattern allowed us to divide up the functionality into multiple states. For example, the ADD_ROW state adds data from the table into the appropriate Column, then transitions to the next state. The program ends when the SUCCESS state is reached. This pattern was useful for controlling the flow of the program.

MAXIMUM GRADE EXPECTED

The maximum grade we are expecting is 11 marks. That would be 10 for the initial project and one bonus mark for using state machines.