Final design document

Patrick Wang
20837029

Overview

My program employs the Model View Controller design architecture. I might've added a bit too much detail, but the first paragraph of each section provides enough to get a general overview. Also, I forgot to explicitly mention in my initial plan of attack that I implemented syntax highlighting for my project.

Model

The Model abstract class represents the state of the editor. It has a method to add a Controller to itself, as well as a method to add Views to itself. It has a method to receive a vector of Actions from its Controller. It can update and display its Views with several different update methods. Model has a pure virtual method *run* that performs the process of repeatedly receiving commands from Controller and then updating Views. Model also contains methods to set and get its exit status, which is used by the main function to know when the editor is exiting.

VmState is a child class of Model that defines the *run* method. It has unique pointers to instances of various classes representing different aspects of the editor. These classes are AbstractClipboard, AbstractHistory and AbstractEditorState. VmState also has a vector of unique pointers to AbstractFileState objects. It also has a normal pointer to the AbstractFileState object that represents the file that is currently being edited.

The AbstractFileState abstract class represents an individual file that is opened by the editor. It has methods to get the read permissions of the file and set the allowed behaviour of the editor for the file. It has many pure virtual functions that represent different operations that can be made on the file, including adding or removing text, moving the cursor, getting the current cursor position, and saving the file.

FileState is a child class of AbstractFileState. Privately, FileState represents the contents and cursor position of its file through a unique pointer to an AbstractFileContent object. The AbstractFileContent abstract class declares many pure virtual methods for modifying or getting information from the contents of the file and moving the file cursor, but these methods are more specific than those of AbstractFileState and make relatively small changes. The FileState class uses these specific, smaller methods to implement the broader methods that are declared by AbstractFileState. FileState's unique pointer to AbstractFileContent is initialized to be a unique pointer to a ReverseGapBuffer object.

ReverseGapBuffer is a child class of AbstractFileContent. It uses a gap buffer data structure to store the contents of a file, with each line being stored as an individual element in one of two vectors. The first vector represents the text before the cursor, and it stores the lines normally.

The second vector represents the text after the cursor, and it stores the lines in reverse. Methods that change the content of the file or move the cursor involve adding or removing text from these two vectors or moving text between them.

There is also a CursorPos struct that represents the position of a cursor. It has a lineIndex and charIndex field to store the coordinates. This struct is shared among all classes that need to get or set a cursor position.

The AbstractClipboard abstract class represents the editor's clipboard. It provides a pure virtual method for accessing the contents of the clipboard as a reference to a vector of strings, as well as pure virtual methods for clearing and appending to the clipboard. Clipboard is a child class of AbstractClipboard, and it privately represents the contents of the clipboard as simply a vector of strings.

The AbstractHistory abstract class represents the history of undoable actions done on a file. These actions can then be accessed for performing undo. The class provides pure virtual methods for adding, getting, and removing the latest undoable action in the history. It also provides functionality for redoable actions, but those did not end up getting used in my program. History is a child class of AbstractHistory. It privately represents the history of undoable actions as a stack, with the latest actions being pushed to the top of the stack. It also has a stack for redoable actions, and can move actions between the stacks. Again, this redo functionality is not used.

The AbstractEditor abstract class represents the remaining aspects of the editor that are not covered by the other classes. These include the command bar content and cursor, the last string or character that was searched, the search direction, the text that is overwritten in replacement mode, and whether a file has already been changed in insert mode. It provides pure virtual methods to get and set all of these aspects. EditorState is a child class of AbstractEditorState. It privately represents its aspects with simple data types. The command bar content is represented by a string, the last search terms are represented by a string and a character, and so on and so forth.

View

The View abstract class helps display a certain part of the editor. It provides several pure virtual methods to update itself, and these methods take different inputs depending on the intended behaviour. It also provides a pure virtual method to display something to the screen.

FileView is a child class of View. It helps display a file that is opened by the editor. Privately, it has normal pointers to a VmState object representing the editor and a AbstractFileState object representing the file it is displaying. FileView only displays its file when it is the same as the VmState object's current file. Privately, FileState also has unique pointers to instances of the Colorer and SyntaxParser class, and it uses them to display different colors for different parts of the file content. FileState defines methods to change the display in different ways. It has a

method to simply change the position of the cursor, and the method itself will call another update method if scrolling occurs. It also has a method to simply update the entire display, as well as methods to update certain lines or blocks of text. For these methods, FileState will access the contents of its file through the AbstractFileState object which can return a constant reference to a vector of strings representing the file. The update methods of FileState then go through the relevant parts of the vector. Each line is passed to the SyntaxParser, which identifies a token at startIndex, and returns information about the type of that token (keyword, comment, etc.) and its length. The type information is then passed to the Colorer which then returns a color for that token. The update methods then display the token with the given color. This process continues, with startIndex being incremented, until the end of the line is reached.

There is a SyntaxItem enumeration for the different types of token defined for this project. There is also a ParseItem struct representing the type and length of a token.

The SyntaxParser abstract class takes in a line and a starting index, and returns a ParseItem representing the token that starts at that index. It also provides a method to update its information given the contents of a file, which does nothing by default. CppParser is a child class of SyntaxParser and is designed for syntax highlighting in C++ files (.h and .cc). Privately, it has a map that goes from single characters to C++ keywords that begin with those characters. It also has a vector of regex objects to help identify preprocessor directives, and a vector of identifiers that are present in the file which can be updated. These fields are used in CppParser's implementation of the *parse* method. NormalParser is another child class of SyntaxParser and is designed for regular files. For any given line, NormalParser's implementation of parse simply returns a ParseItem with the same type and the length of the line.

The Colorer abstract class simply returns color for a given SyntaxItem enumeration. VmColorer is a child of Colorer, and it has a private map that goes from SyntaxItem enumerations to ncurses colors.

CommandBarView is another child class of View. It helps display the command bar of the editor. It has methods to update the contents of the command bar as well as the cursor in the command bar. Privately, it has a normal pointer to a VmState object representing the editor. When it updates it can get the cursor position in the file from the VmState's AbstractFileState and it can get the command bar content from the VmState's AbstractEditorState, and then display this information.

Controller

The Controller abstract class takes care of the input for the editor. It provides a method that returns a series of pointers to Action objects. The method return should be triggered by some user input. CursesKeyboard is a child class of Controller. It has unique pointers to InputParser and CursesKeyBinding objects. Privately, it represents the input buffer as a vector of integers.

There is a Mode enumeration for the different types of modes that the editor can be in.

The InputParser abstract class takes in a buffer of integers representing ncurses keystrokes and returns a vector of Action objects. VmParser is a child class of InputParser. Privately, it has four vectors of unique pointers to Action objects. Each vector holds actions for a different mode (normalActions, insertActions, etc.). It has a field indicating which mode the editor is in, and based off of that it loops through the corresponding vector of Action objects. For each Action object it calls the Action's *matches* method to see if the input it received trigger that Action. If it does it add the Action to the return vector. VmParser itself does not implement any of the behaviour of the Actions, it simply returns the ones that match a given input. It also has fields to track multipliers at the start of the input and the last normalAction performed.

The CursesKeyBinding abstract class represents the mapping of a keyboard by taking in an ncurses keystroke and returning an ncurses keystroke. NormalBinding simply returns the same keystroke, and so each key is mapped to itself and no change is made.

Actions

The Action abstract class represents the behaviour of the editor in response to a certain input. It provides pure virtual methods to indicate whether it matches a given input or partially matches a given input. For matches, it returns a matchInfo struct which provides additional information indicating any potential need to switch modes or turn on the command bar cursor. It also provides an *action* method that takes in a pointer to a VmState object. This is so that the action has access to the VmState's public methods and can thus perform its own behaviour through method calls.

There are many child classes of the Action class. Overall, these subclasses are split into four groups for the four different modes of the editor. Each subclass also represents a group of similar commands. As an example, the Search class represents all actions that search a file for a given input. Some Action subclasses also provide helper functions for other subclasses to use. As an example the Movement and QuickMovement classes provide helper functions for the ChangeText class for commands that change the text in a given motion.

UML

Not much changed with the overall structure of my UML. The classes and the relationships between them mostly stayed the same. I got rid of the AbstractCursorState abstract class since it was redundant. I already had enough information about the cursor in AbstractFileState and AbstractEditorState. I also removed the KeyBuffer class that's owned by CursesKeyboard, since it was not needed. I found that a vector of integers was easier to work with than strings (KeyBuffer transformed a series of integers into a string). I also changed the subclasses of Action so that more similar commands could be grouped together, thus increasing cohesion. Other than these changes to the classes, I added more methods to a lot of classes to increase their functionality (this is especially true in AbstractFileState, FileView and Action)

Design

The MVC architecture was used for the overall structure of my program. Even though we needed to use it according to the project specifications, I still think it's the best option since it allows for lower coupling and higher cohesion, and thus the distinct parts of the program can be modified relatively easily. This architecture employs the Observer pattern.

Model

The Model abstract class only provides implementation for getting Actions from its Controller and updating its Views. These methods, as well as the structure of the Action object, are fundamental to the program and unlikely to change. The methods do not rely on the implementation of View or Controller, thus providing low coupling. Model is also not in charge of creating instances of View or Controller and as such it can deal with View and Controller as abstract classes, following the Dependency Inversion Principle (DIP).

The VmState child class follows the Single Responsibility Principle (SRP), since it only brings together the different aspects of the editor in the implementation of the main *run* method. Also, since VmState only has unique pointers to the abstract classes representing the different aspects of the editor, it follows the DIP. The implementation of these abstract classes can easily be swapped out, thus VmState also follows the Open/Closed Principle (OCP). Furthermore, grouping the main aspects of the editor into different classes allows the individual classes to follow the SRP more closely. It also allows for low coupling between the classes, since they have isolated responsibilities and thus don't need to know anything about each other, and high cohesion, since the classes are organized by type of responsibility.

The AbstractFileState abstract class provides a wide range of methods representing actions that can be made on the file. These methods are very general and have high cohesion (they all act on the file). They are not specifically designed for any one type of use, thus encouraging their reuse when implementing different behaviours. The Non-Virtual Interface (NVI) idiom, and by extension the Template Method pattern, is not used for these methods as there is no check that needs to be made on their inputs. Furthermore, there are so many methods that implementing NVI would be extremely tedious (and I didn't have that much time).

The FileState child class represents the content of the file with an AbstractFileContent class, thus it follows DIP. This also allows FileState to swap out one implementation of AbstractFileContent for another, perhaps in a situation where certain of its methods need to have a faster implementation. Thus FileState follows the OCP.

The AbstractFileContent abstract class provides many methods for modifying the contents of a file and moving the file cursor. However, these methods are much more specialized and do very small and straightforward tasks. As such, they have general applicability and can be reused to do more complicated tasks, as is the case in FileState. Furthermore, it's easier to optimize these

individual, smaller methods to meet some guideline, as well as to provide no throw guarantees in order to avoid throwing in the places that use these methods. The ReverseGapBuffer child class optimizes the implementation of these methods to ensure that operations for moving the cursor by a single position or for adding/removing single characters are faster, since it is assumed that these operations will be the most common ones for users of the editor.

The remaining abstract classes owned by VmState (AbstractClipboard, AbstractHistory, AbstractEditorState) and their child classes are quite straightforward. Each abstract class provides an interface to interact with a specific part of the editor, and so they follow the SRP. This also means that the methods and fields of a single class are all highly related to each other, and so there is high coupling. Furthermore, none of the abstract classes are designed to need any information from another part of the editor, and so there is no interaction between them and thus there is low coupling. A change to one aspect of the editor does not affect the others.

View

The View abstract class simply provides an interface of methods that other classes can use. This interface is implemented by both child classes of View, FileView and CommandBarView, so that calling either subclass results in proper, well defined behaviour. As such, the child classes can be used in any place that requires a View and so they follow the Liskov Substitution Principle (LSP).

The FileView child class is responsible only for displaying a single file, and so it follows the SRP. It itself is not dependent on any particular screen size, and thus can theoretically be dynamically resized. It is not dependent on any other class FileView contains a pointer to a VmState object, as well as a AbstractFileState object, and uses their public methods in its implementation. Thus there is some coupling between the classes. This is mainly for the sake of convenience, so that FileView has a less tedious way of accessing the content of its file. One can imagine changing the update method of View to accept a reference to the contents of a file, but that would be tedious to call and might not be desirable for other types of View. Also, FileView only needs a pointer to an AbstractFileState object to compare with the pointer to VmState's current open file, to see if it can update its display when there are multiple files. As such, any changes to the implementation of AbstractFileState do not actually affect FileView. Ultimately, this decision to introduce some coupling between these classes is mainly for convenience. Also, VmState is at the core of the program and its structure is unlikely to change. So, it is unlikely that this coupling will present any issue down the line.

FileView also has unique pointers to the Colorer and SyntaxParser abstract classes, and as such follows DIP. The Colorer class maps each type of token to a color, while the SyntaxParser identifies tokens based on a given syntax. As such, it is likely that one might want to change these properties before or even during the runtime of the program. In fact, this already happens to some extent in the main function of the program, as it first scans the ending of the file name before either constructing a NormalParser or CppParser for FileView. Since the child classes of

Colorer and SyntaxParser implement their virtual methods for proper behaviour, they follow LSP and thus can be swapped interchangeably. Additionally, the Colorer and SyntaxParser classes both follow the NVI idiom, and by extension the Template Method pattern, since their public methods first check that their inputs (SyntaxItem, string with startIndex) are valid before calling their private pure virtual methods, which are defined by the child classes. Any child classes that may be implemented cannot change the initial check.

The CommandBarView child class is responsible only for displaying the command bar, and so it follows the SRP. It has a pointer to a VmState object, so there is some coupling. However, this is mainly for convenience so that CommandBarView can easily access the contents of VmState's command bar. Once again, the structure of VmState is unlikely to change.

Controller

The Controller abstract class simply provides an interface that the Model can call to receive a series of Action objects triggered by some user input. It follows the NVI idiom by declaring a private pure virtual *action* method that is called by the public *getAction* method. While getAction currently doesn't do any additional thing other than call action, this Template Method pattern could be used to check if the returning Action objects might cause some sort of problem.

The CursesKeyboard child class is solely responsible for receiving raw user input from the keyboard, and then passing it through the CursesKeyBinding and InputParser objects that it owns. As such it follows the SRP. Furthermore it is not dependent on any instance of a particular subclass of CursesKeyBinding or InputParser, and thus it can treat them as abstract classes. So, CursesKeyboard follows OCP and DIP, and this allows for low coupling. It is likely that some user might want to change the bindings of the keyboard to different inputs, or implement a different set of textual commands that lead to different behaviour.

The InputParser abstract class follows the NVI idiom, and by extension the Template Method pattern, since its public *parseAction* method calls its private pure virtual *action* method for its return value. This pattern allows the parseAction to first check the user input for any sequences or keys that are simply not allowed by the program or may cause issues with the *action* method itself. This check cannot be overwritten by any child class, ensuring some sequences of inputs are never accepted. The VmParser child class has four vectors of different Action objects (for the four different modes) and it is not dependent on any instance of a particular subclass of Action. As such, it treats its Action objects as abstract Action objects. After receiving an input, it loops through all the Actions in the vector corresponding to a given mode, and it does not implement any of the behaviour triggered by a given Action. So, VmParser follows OCP and DIP. New Actions objects that might be created can easily be added to the VmParser's vector of Action objects without affecting its implementation or correctness.

Action

The Action abstract class provides an interface of methods that InputParser can call to determine if a given Action should be sent to Model. Action also follows the NVI idiom, since its public *performAction* method calls its private pure virtual *action* method. This allows performAction to first verify that the VmState input it receives is in a valid state for the Action to perform its behaviour. In my program there are many child classes of Action. Each subclass is responsible for a single type of behaviour, such as Movement or UndoRedo, and so the Action subclasses follow SRP. This also means there is low coupling between the Action subclasses and very high cohesion within them, since they're grouped by type of behaviour. Of course, there is still some coupling between some Action subclasses, which is necessary for a vim-like program. As an example, the implementation of the command that deletes text in any given motion is placed in the ChangeText subclass. However, the 'any motion' part of the command means that the implementation must also know what the other movement commands, for both determining whether a given input is valid and also to avoid the code duplication of movement. As is, the coupling between the Action subclasses is as low as it can be.

Resilience to Change

Since every class in my program follows the DIP when applicable, changes in implementation can often be accommodated easily. There should be no change to correctness (at least on my part) and very little recompilation (since my program generally has low coupling).

The VmState is solely dependent on the interfaces of abstract classes. As such, any implementation of these abstract classes can be swapped out if necessary. For AbstractFileState, if the user needs a file that is more optimized for making quicker changes to large amounts of text at a time, then they can simply create an optimized subclass that implements AbstractFileState's interface and pass it to VmState's constructor. Even for FileState, which is the current subclass of AbstractFileState, it represents the contents of a file with an AbstractFileContent object. Currently, that field is initialized with a ReverseGapBuffer subclass, which represents a gap buffer data structure. However, perhaps the user may want to use a different data structure with different runtime and memory benefits. If so, they can simply create the structure and implement the interface of AbstractFileContent, and pass that subclass to FileState's constructor.

This process of changing the underlying implementation works for all the fields of VmState (AbstractFileState, AbstractClipboard, AbstractHistory, AbstractEditorState), each representing a different aspect of the editor.

Of course, if the user wanted to change the actual behaviour of these abstract classes, or wanted to add more functionality with methods that are perhaps more complicated, that might require some work to actually change the abstract classes and their child classes. However, this is changing the functionality of the editor which is fundamental to the program, and so some work should be expected. Nonetheless, from the way I've designed the abstract classes owned by VmState, these changes in functionality should require minimal work, especially in the case of AbstractFileState and its child class FileState. The AbstractFileContent class provides many

small and specialized methods all related to modifying the contents of a file and moving the file cursor. The methods follow the SRP in the sense that each one takes care of a single task in an optimized fashion. Since FileState represents its file with an AbstractFileContent object, it can use these smaller methods to build more complex methods easily and with less worry about correctness. So, the user can add functionality to FileState easily. Of course, FileState and AbstractFileState themselves also provide many methods that, while more general than the methods of AbstractFileContent, are still quite specialized and cover the bulk of operations one might need to do on a file. As such, for any new file modification functionality that a user might want to implement, they should already be able to implement it themselves using the methods of AbstractFileState, which should be exception safe and optimized.

The FileView child class also allows for change, especially with regards to syntax highlighting, which is some that the user will likely want to change. FileView follows DIP, since its colouring of text is solely dependent on the abstract classes Colorer and SyntaxParser. As such, if the user wanted to add syntax highlighting for another language like Python, they could simply make another subclass that implements the interface of SyntaxParser. Then in the main function of the program they could add a condition that checks for Python files (.py?) and if the condition passes, the main function will pass the new subclass to the FileView constructor. If the user wants to change the colouring of different syntax items, they can make another subclass of Colorer and pass that subclass to the FileView constructor. In fact, FileView being only dependent on the interfaces of Colorer and SyntaxParser means that the program can easily be changed to allow the user to switch colors and language syntax dynamically during runtime.

The CursesKeyboard child class also allows for change with regards to key bindings. CursesKeyboard follows DIP and only depends on the CursesKeyBinding abstract class for receiving input. As such, a user can simply create a new subclass of CursesKeyBinding and pass it to the CursesKeyboard constructor to change key bindings. This also means that the program can be changed to allow the user to change key bindings dynamically during runtime.

Now, the thing that a user is very likely going to want to change or add to are the commands for the editor. The user may want to change the actual behaviour of a certain command, or they may want to add a new command. Luckily, it should be very easy to perform these changes to my program. The key is that each subclass of Action implements a method that takes in a pointer to a VmState object. This is the method that is called by the Model on itself in the MVC loop. As such, in this method the Action subclass has access to all the public methods provided by VmState, including all the functionality provided by the different aspects of the editor. Therefore, the way I've designed my program is that the Model, View and Controller implement none of the behaviour of any of the editor commands. It is then the responsibility of each subclass of Action to implement its own behaviour, as well as the methods that match it to a certain set of inputs. This allows for low coupling, since implementations of commands aren't depended on by the rest of the program, and high cohesion, since the subclasses can be organized by type of command. I intentionally designed my program like this from the start, since I think it's reasonable to say that commands are the most likely to change or be added to out of everything in the program.

So suppose a user wanted to add a new command. They could create a subclass of Action. In this subclass, they would implement the actions matches and partialMatches that essentially return whether a certain series of character inputs (ncurses characters) will trigger or might trigger that command. Then they would provide an implementation of Action's private virtual *action* method. In this method, they would receive a pointer to a VmState as input. Then, the user could use the entire functionality provided by VmState and its fields (which is very extensive, as previously shown for AbstractFileState) to modify the VmState. From the way I've designed the component classes of VmState, the user should have enough building block methods to implement the desired functionality. Finally, the user can simply add this new subclass to the vector of Actions in VmParser (this can be done through a method call). As such, none of the rest of the program is affected by this change. This should also maintain low coupling and high cohesion, but that is solely dependent on the user.

Therefore, a user can add a command that is triggered by any series of keystrokes they want, and that can have any behaviour that is made possible by the extensive interface of VmState. It is up to them to implement it properly. This change should not affect the rest of the program.

Questions

Multiple Files

To support multiple files in my program, the VmState class contains a vector of AbstractFileState objects and has a field indicating which object is the current file. As such, you can have multiple AbstractFileState objects corresponding to multiple files. A method can be added to VmState to switch the current file. As previously mentioned, new commands for switching between the files can also easily be added. All my current commands act on the current file of VmState, which should be their intended behaviour for multiple files anyway.

As for how these multiple files should be displayed, the way I've designed my program presents two options.

One option is to create a FileView object for each individual file. Currently, when a FileView's update method is called, it will only update if it is the current file (it has access to VmState to get this information) which is good. I can also easily add getter and setter methods for the sizes of each FileView, so that they can be resized dynamically during runtime. Furthermore, for the layout of the FileView objects, it might be a good idea to use the Decorator pattern. My current program doesn't not allow for this since the FileView objects do not have pointers to other View objects, but it shouldn't be too much work to implement it. The actual functionality of the FileView objects wouldn't change, since visually they are solely dependent on the size fields of the FileView itself. As such, resizing the FileView should not change its correctness.

Another option is to only have one FileView object, and to change its AbstractFileState field when switching between files to always point to the current file. All this requires would be to add

a getter and setter function for the AbstractFileState pointer of FileView, which is very easy. The remainder of the program should stay exactly the same, since the Action objects only act on the current file, and new Action objects to switch between files would switch the display as well. The only downside to this option is that only one file can be displayed at once.

Read/Write Permissions

The AbstractFileState class has a boolean field indicating whether the file it corresponds to is read-only (this field is set in the constructor). Furthermore, it also owns an instance of the PermissionBehaviour abstract class, thus following DIP. As such, the subclasses of PermissionBehaviour, which represent the programs behaviour in response to read-only and write-only permissions, can be swapped interchangeably to implement different behaviours (this can occur during runtime).

The AbstractFileState already implements the *saveFile* method, and so it can check its own read-only field and its PermissionBehaviour field before allowing the user to save. In terms of preventing edits, the program is designed so that any edits made to AbstractFileState are only displayed if its *updateContent* method is called. Currently, the *updateContent* method is public and overridden by the FileState child class. However, it is very easy to make updateContent a private virtual method, and to add a public method that calls *updateContent*, thus following the NVI idiom. Then, the public method can check the read-only field and the PermissionBehaviour field before allowing updateContent to be called. This would ensure that the right PermissionBehaviour is enforced for both saving and editing files for all child classes of AbstractFileState.

The rest of the program is not affected, including the Action objects that operate on VmState. The changes made by any Action object is only displayed after calling updateContent. If updateContent is controlled by the read-only permissions, there is nothing an Action can do, since it only has access to VmState's public methods.

Final Question

If I had a chance to start over, the only thing I would do differently would be to start my project earlier. This would have given me more time to better plan and design my program. I also could have reviewed the course notes more to see if there were any abstractions I could've used to improve my program. However, for the time that I spent on this project I can say that I'm pleased with what I made. Of course, I'm sure there are many areas in my program that could've had better design, but that all comes with time.