Assignment 1: Design

Jeremiah Griffin

3 Feb – Winter 2017

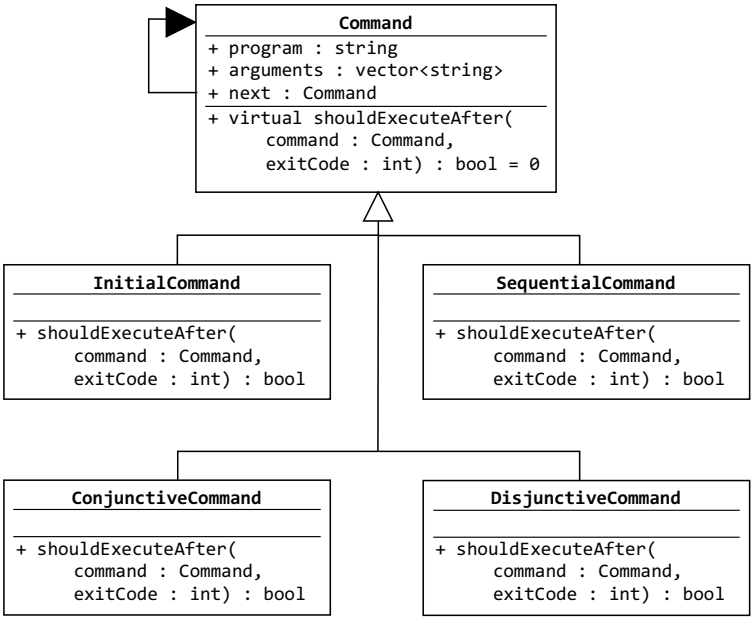
1 Introduction

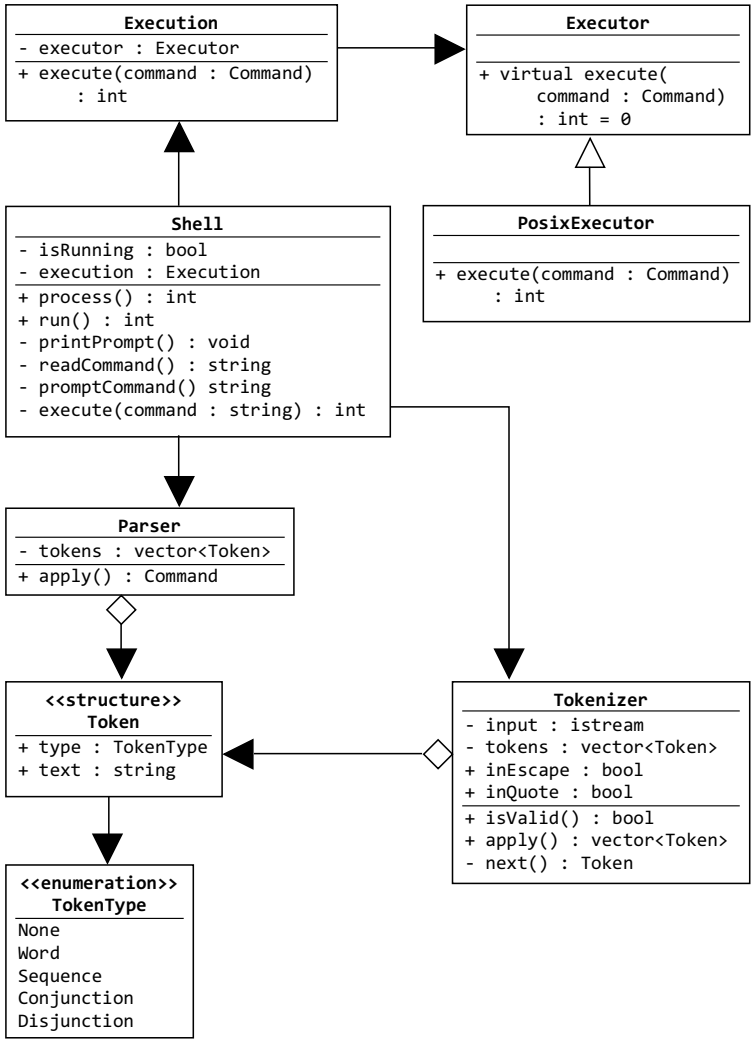
In this project I will design a basic POSIX command shell using the composite and strategy patterns with a strong emphasis on object oriented design principles and correctness. This design separates the semantic structure of commands from its model of execution and allows the user a high degree of composability while simultaneously leaving the design of the system open for future extensions wherein the developer might create a more feature-complete shell. The classes set forth in this design use language constructs intended to ensure that sensible invariants are necessarily preserved at all times, making it largely impossible to create an invalid object graph in the domain of the design.

The operation of this design is in a loop, the body of which is as follows:

1. The `Shell` instance prompts the user for a command.
2. The provided command is tokenized by the `Tokenizer` class.
3. The command tokens are parsed into one or more `Command` instances using the composite pattern to represent chained commands and their conditions.
4. The `Command` instance is given to an `Execution`.
5. The `Execution` processes each `Command` in the composition using its `Executor`, which is instantiated according to the underlying operating system and performs the necessary platform-specific operations to execute the given command.

2 Diagram


Command
+ program : string
+ arguments : vector<string>
+ next : Command
+ virtual shouldExecuteAfter(
command : Command,
exitCode : int) : bool = 0
InitialCommand
+ shouldExecuteAfter(
command : Command,
exitCode : int) : bool
SequentialCommand
+ shouldExecuteAfter(
command : Command,
exitCode : int) : bool
ConjunctiveCommand
+ shouldExecuteAfter(
command : Command,
exitCode : int) : bool
DisjunctiveCommand
+ shouldExecuteAfter(
command : Command,
exitCode : int) : bool


Execution
- executor : Executor
+ execute(command : Command)
: int
Executor
+ virtual execute(
command : Command)
: int = 0
Shell
- isRunning : bool
- execution : Execution
+ process() : int
+ run() : int
- printPrompt() : void
- readCommand() : string
- promptCommand() string
- execute(command : string) : int
PosixExecutor
+ execute(command : Command)
: int
Parser
- tokens : vector<Token>
+ apply() : Command
<<structure>>
Token
+ type : TokenType
+ text : string
Tokenizer
- input : istream
- tokens : vector<Token>
+ inEscape : bool
+ inQuote : bool
+ isValid() : bool
+ apply() : vector<Token>
- next() : Token
<<enumeration>>
TokenType
None
Word
Sequence
Conjunction
Disjunction

3 Classes

3.1 `Shell`

This is the class which presents a user-friendly interface and provides the point of interaction for accepting and executing user commands. It is responsible for instantiating an appropriate `Execution`, printing command prompts, accepting commands, tokenizing and parsing command strings into `Command` instances, and offering commands for execution.

The command string input process allows for multiline commands. If a command ends with a backslash or in the middle of a quoted string, the continuation prompt is printed and another line of input is accepted, appended to the command string. If the previous line terminated with a backslash, the next line is joined to the command string directly. If the previous line was terminated in the middle of a quoted string, the next line is joined with a line terminator. This process may be repeated any number of times.

3.1.1 Interface

```
class Shell
{
public:
    bool isRunning() const noexcept;

    int process();
    int run();

private:
    bool _isRunning{false};
    Execution _execution;

    void printCommandPrompt();
    void printContinuationPrompt();
    std::string readCommand();
    std::string promptCommand();

    int execute(const std::string& string);
};
```

3.1.2 Members

- `isRunning`: Whether or not the shell is running. This will be set to false when the `exit` command is received.
- `execution`: The execution instance to used for executing commands.

3.1.3 Methods

- `process`: Prompts for, reads, and executes a command.
- `run`: Repeatedly calls `process` as long as `isRunning` is true.
- `printCommandPrompt`: Prints the prompt to begin a command.
- `printContinuationPrompt`: Prints the prompt to continue a command.
- `readCommand`: Reads a command string from the standard input, prompting for continuation as necessary.
- `promptCommand`: Prints for and reads a command string.
- `execute`: Tokenizes, parses, and executes the given command string.

3.2 `Token`

This structure represents a lexical token within a command string, such as a command word or chaining connective.

3.2.1 Interface

```
struct Token
{
    enum class Type
    {
        None,
        Word,
        Sequence,
        Conjunction,
        Disjunction,
    };

    Type type;
    std::string text;
};
```

3.2.2 Enumerations

- `Token`: The types of tokens that may appear in a command string.
  - `None`: Represents the end of a token stream.
  - `Word`: A space-delimited word or quoted grouping of words.
  - `Sequence`: A sequenced command delimiter (;).
  - `Conjunction`: A conjoined command delimiter (&&).
  - `Disjunction`: A disjoined command delimiter (||).

3.2.3 Members

- `type`: The enumerated type of the token.
- `text`: The body text of the token.

3.3 `Tokenizer`

This class accepts a stream and transforms it into a sequence of `Token` instances through lexical analysis. The classifications used in this analysis are represented with the following regular expressions, in the order of descending precedence, where the \s sequence represents a white-space character and each expression is surrounded by any number of white-space characters on either side:

```
Sequence = ;
Conjunction = &&
Disjunction = \|\|
DirectWord = [^\s"\\]|\\.
QuotedWord = "([^"\\]|\\.)"
Word = (DirectWord|QuotedWord)+
```

During this process, escaped sequences are processed and replaced with the appropriate character, as defined in the following mapping:

Sequence	Substitution
\a	Bell
\e	Escape
\n	Line feed
\r	Carriage return
\t	Tab
char	*char*

The last substitution in the table above allows for the insertion of quotes, backslashes, and other special characters as elements of a word.

3.3.1 Interface

```
class Tokenizer
{
public:
    explicit Tokenizer(std::istream& input);

    const std::vector<Token>& tokens() const noexcept;
    bool inEscape() const noexcept;
    bool inQuote() const noexcept;
    bool isValid() const noexcept;

    const std::vector<Token>& apply();

private:
```

```
    std::istream& _input;
    std::vector<Token> _tokens;
    bool _inEscape;
    bool _inQuote;

    Token next();
};
```

3.3.2 Members

- `input`: The input stream to tokenize.
- `tokens`: The produced sequence of tokens.
- `inEscape`: Whether or not the tokenization terminated during an escape sequence.
- `inQuote`: Whether or not the tokenization terminated during a quote sequence.

3.3.3 Methods

- `isValid`: Returns a value indicating whether or not the tokenization terminated normally.
- `apply`: Repeatedly tokenizes until the end of the input stream is reached, returning the produced tokens.
- `next`: Obtains the next token from the stream. If there are no more tokens available, the returned token will have the `None` type.

3.4 Parser

This class accepts a sequence of tokens and transforms it into a chain of one or more commands, provided that the token stream meets the grammatical requirements of a command, as defined in the following recursive grammar notated in extended Backus-Naur form, using the expressions defined in the language of the `Tokenizer` class:

```
Command = SimpleCommand, [ ChainedCommand ] ;
SimpleCommand = Word, { Word } ;
ChainedCommand =
    SequentialCommand |
    ConjunctiveCommand |
    DisjunctiveCommand ;
SequentialCommand = Sequence, Command ;
ConjunctiveCommand = Conjunction, Command ;
DisjunctiveCommand = Disjunction, Command ;
```

3.4.1 Interface

```
class Parser
{
public:
    explicit Parser(const std::vector<Token>& tokens);

    std::unique_ptr<Command> apply();

private:
    const std::vector<Token>& _tokens;
};
```

3.4.2 Members

- `tokens`: The sequence of tokens to parse as a command.

3.4.3 Methods

- `apply`: Parses the token sequence according to the defined grammar and returns the resultant command. If the sequence is not well-formed, a null pointer is returned.

3.5 Command

This class serves as the abstract base class in the composite pattern of the command structure. It represents a polymorphic command to be executed.

3.5.1 Interface

```
class Command
{
public:
    std::string program;
    std::vector<std::string> arguments;
    std::unique_ptr<Command> next;

    virtual ~Command();

    virtual bool shouldExecuteAfter(const Command& command,
            int exitCode) const noexcept = 0;
};
```

3.5.2 Members

- `program`: The name of the program to execute.
- `arguments`: The arguments to pass to the program upon execution.

- `next`: The next command to execute within the chain, if any.

3.5.3 Methods

- `shouldExecuteAfter`: Returns true if this command should be executed after the given command, which exited with the provided code upon completion.

3.6 `InitialCommand`

The first command within a command chain.

3.6.1 Interface

```
class InitialCommand : public Command
{
public:
    virtual ~InitialCommand();

    virtual bool shouldExecuteAfter(const Command& command,
            int exitCode) const noexcept override;
};
```

3.6.2 Methods

- `shouldExecuteAfter`: Returns false, as an initial command should never be executed after another command.

3.7 `SequentialCommand`

A command to be executed unconditionally within a command chain.

3.7.1 Interface

```
class SequentialCommand : public Command
{
public:
    virtual ~SequentialCommand();

    virtual bool shouldExecuteAfter(const Command& command,
            int exitCode) const noexcept override;
};
```

3.7.2 Methods

- `shouldExecuteAfter`: Returns true, as a sequential command is always executed.

3.8 ConjunctiveCommand

A command to be executed if the preceding command exited successfully with a zero exit code.

3.8.1 Interface

```
class ConjunctiveCommand : public Command
{
public:
    virtual ~ConjunctiveCommand();

    virtual bool shouldExecuteAfter(const Command& command,
            int exitCode) const noexcept override;
};
```

3.8.2 Methods

- `shouldExecuteAfter`: Returns true if the exit code is zero.

3.9 DisjunctiveCommand

3.9.1 Interface

```
class DisjunctiveCommand : public Command
{
public:
    virtual ~DisjunctiveCommand();

    virtual bool shouldExecuteAfter(const Command& command,
            int exitCode) const noexcept override;
};
```

A command to be executed if the preceding command exited unsuccessfully with a nonzero exit code.

3.9.2 Methods

- `shouldExecuteAfter`: Returns true if the exit code is nonzero.

3.10 Execution

This class represents the algorithm for executing commands in the strategy pattern. It attempts to execute all commands in a chain, using the results of the preceding command to determine whether the subsequent command should truly be executed. Commands that indicate that they should not be executed are skipped, treating the last executed command as the preceding command for the next. It uses the `shouldExecuteAfter` method on each command to determine whether it should be executed and its internal `Executor` instance to perform the execution of commands.

3.10.1 Interface

```
class Execution
{
public:
    explicit Execution(std::unique_ptr<Executor>&& executor);

    const Executor& executor() const noexcept;
    Executor& executor() noexcept;

    int execute(const Command& command);

private:
    std::unique_ptr<Executor> _executor;
};
```

3.10.2 Members

- `executor`: The executor to perform the execution of commands with.

3.10.3 Methods

- `execute`: Executes the given command structure using the algorithm given above. If at any point a command with the program name "exit" is given and meant to be executed, an exception is thrown, intending to signal to the caller that no more commands should be processed and the shell should terminate. Returns the exit code of the last command executed.

3.11 Executor

This class serves as the abstract base class in the strategy pattern of the execution algorithm. It represents a polymorphic function to perform the actual execution of a single command.

3.11.1 Interface

```
class Executor
{
public:
    virtual ~PosixExecutor();

    virtual int execute(const Command& command) = 0;
};
```

3.11.2 Methods

- `execute`: Executes the individual command given and returns its exit code.

3.12 `PosixExecutor`

This class implements the execution algorithm by executing commands through POSIX system calls.

3.12.1 Interface

```
class PosixExecutor : public Executor
{
public:
    virtual ~PosixExecutor();

    virtual int execute(const Command& command) override;
};
```

3.12.2 Methods

- `execute`: Executes the individual command given and returns its exit code.

4 Coding Strategy

I am working without a partner, thus there is no per-person division of work. Instead, I will describe the order in which I will implement the components of the system. My goal is to have a reasonably functional program after each stage.

1. `Command` class
2. `InitialCommand` class
3. `SequentialCommand` class
4. `ConjunctiveCommand` class
5. `DisjunctiveCommand` class
6. `Token` structure
7. `Tokenizer` class
8. `Parser` class
9. `Executor` class
10. `PosixExecutor` class
11. `Execution` class
12. `Shell` class

5 Roadblocks

The primary roadblock that might be encountered in this design, within the scope of the assignment, is the implementation of command associativity or grouping with respect to conjunctive, disjunctive, and sequential command chains. This problem is currently addressed in the design of the execution algorithm, though this solution may prove inadequate, necessitating a change to the composite design for the command structure, introducing a `CommandGroup` composite class and organizing commands into groups during the parsing phase.

A common feature of shells is builtin commands, such as `alias` in bash and zsh. To implement these, the most straightforward approach in this system is to write a new executor type, `BuiltinExecutor`, which performs the processing. However, this would lead to problems when builtin commands are chained with external commands. To solve this, a system would need to be created to select an executor on a per-command basis. This might be accomplished using a set of `Executor` instances in the `Execution`, which then delegates work to the appropriate executor depending on the nature of each `Command`.

Another feature that is not directly addressed in this specification is command comments, which are necessary for this assignment. A likely point of implementation for this feature is within the `Tokenizer` class, whose language could be extended to discard all characters after a comment token.