# Team: P37

Throughout the program, we used chain of command OOP to ensure that classes only had 1 purpose, functionality and use. We used an abstract class to define the base of the builder, runner and command classes, so that all implementations of an abstract class would have the same method arguments and returns, allowing for a predictable and easy to develop program. The chain of command would be as follows: Parser -> visitor -> builder -> runnable (command).

Firstly, the shell parses the given command line into a builder using ANTLR and our runner visitor logic. This is where we had our first additional feature. We created an auto importer to automatically import all commands from the src/commands/ folder, then read their included FlagSpecification to inform the parser what flags (and their names and types) are allowed for each command, along with the command name. This allows the parser to  parse the flags and catch invalid ones at the parsing stage, rather than the running stage. Furthermore, it allows future developers to implement the command in the src/commands/ folder, without modifying other parts of the parser or shell.

During visiting, there are 2 different scenarios: visiting commands and visiting subcommands (RunnerVisitor, SubcommandVisitor). The runner visitor is responsible for parsing everything and then building them into runnable objects such as pipes, seq, and commands. The Subcommand visitor is responsible for evaluating everything inside the backquotes, returning a string with the subcommand evaluated and any trailing white space or new lines is removed. This subcommand evaluation is another extension feature in our program, as its complexity allowing for commands such as:
*echo `echo mo`ck_comm`echo and`* to work fully and completely.

Finally, the shell runs the commands, where the command controls the input and output stream of the command. All commands inherit from an abstract class base command, where the run method is overwritten with the logic of the command in question. We also added exception_handled_open, to auto handle exceptions that are thrown when a file is opened, reducing code duplication throughout the commands and tests. We implemented the 'sed' and 'wc' commands, along with command history and escaping sequences allowing for ', " and \ to be interpreted as literals

Regarding testing, each file that was added for a feature would have its own test file. This test file would use mocking to enable the feature to be tested in isolation, without involving any other part of the shell. Systems tests would also be run to ensure that program flow was not altered. Parameterized expand also allowed for units to be tested on a range of erroneous, boundary and normal data, without the need for the duplication of test functions.

We used the SCRUM methodology, with weekly sprints to enable the rapid prototyping of the program. Each feature to be added would be discussed, given a time estimate and a priority, and be added to the Kanban board. The feature would then be implemented by the designated person, then a PR would be reviewed once all github actions passed. The PR would go through the review process before merged into master. Regarding github actions, we had 3 linters, and the following would be run: coverage test, unit tests, mutation tests. As a result of this, all files (except ANTLR generated) had 100% coverage. However with mutation testing, false positives became common so we used it as a general metric rather than a requirement.

## Reviewed submission: 0d3ab9c64a38096d2e6c133e2842ca5aa4a9ac1471b2d8c090c0e51de1fac914

### Design Score: 2

The overall design seems to have had some thought put into it. ANTLR is used for the majority of the parsing, with regex being used for ‘ “ ` parsing (Evaluator.py L110). An attempt had been made to integrate the different quotes into the parser, however the final result of using regex for quote handling and sub-command handling does not adhere to a consistent design pattern, since ANTLR can handle this

Applications inherit from an Application class, however this is not labelled as an abstract class, and inheriting classes do not correctly override the run() method throughout. Changing the run to the signature run(self, arguments: [Argument]) -> str: would better match this.

The design pattern of “Chain of Responsibility” is used here, where the parser returns a tree of commands, where pipe, seq and redirect are handled, which then calls the relevant sub-commands, and commands themselves are then ran. This design has been attempted, however the manual handling of quotations breaks this principle.

### Code Quality Score: 2

The code written has understandable variable and function names, with function headers indicating the types they accept and return. Flake8 tests and static analysis tests both passed. Throughout the project, there is a lack of docstrings (module, class and method) which makes the architecture and implementation difficult to follow. Either documentation in the README file, or documentation using comments and docstrings would greatly help the readability of the code.

Regarding tests, the overall coverage is low, with standout files being shell.py (54%) and applications (82%). Where tests have been written, they rely on project files such as README.md and requirements.md, meaning that the tests do not work stand alone. This could be done using mocking, where mock files with mock actions are used to emulate functions which affect files. Each file and class should have their own tests, that can run using mocked functions without the need for any other classes that call it to be implemented.

The structure of the project could also be improved. Applications.py is a 500 line long file which could be changed into a directory with each application having their own file. The structure of the unit tests could be improved too, each src/ file having a corresponding test/ file. These recommendations would make collaboration easier, since each application being developed would have a new test and application file, rather than having to merge conflict into the Applications.py and tests.py files.

Throughout the Applications.py files, there are functions signatures which do not reflect the base Application class, i.e. cd.run() (L197) returns a string, and doesn’t take a deque as an argument, not reflecting the Application base run() method, going against the OOP principle of inheritance.

### Error Handling Score:  2

Some errors were handled in this program, with file related and value related errors correctly caught and handled. FileNotFound errors were handled, however PermissionError and FileIsADirectory errors were not handled, potentially raising unexpected errors. During parsing, errors are caught as Exception (Converter.py L125). These errors can only be caused by a developer not properly implementing the pipe, hence should have their own type of error.

## Reviewed submission: 7e89febd119dbb7929789ffe75b112eb2f39630d649b9d12316c4906a034f305

### Design Score: 4

This design is complete, understandable, and well implemented. It has good project structure, with well thought out abstractions. An app factory is used to manage the applications and their unsafe versions, where the applications all inherit from a base cmdApp class. However, each of the command implementations of cmdApp inherits from pythons inbuilt ABC, which is incorrect, since the cmdApp class should be inheriting from ABC.

Unsafe applications could be implemented cleaner, since all applications raise an error string so that the app can check if its unsafe or not, then decide to throw the error or put the error string to stdout. A better way to do this would be to throw the error, and have an unsafe wrapper around the application, which will catch and pass on the error to stdout, rather than having an error string within the app.

Pipe, Sequence and Redirection are all written well, with their input streams being set, then the applications being ran.

This is a well thought out inheritance and factory pattern.

### Code Quality Score: 4

One issue with the code is that there are no declared types throughout the program. Return types and argument types should be declared in the function, with a docstring explaining each argument, return and error type. This would greatly help with code readability, since there is no other documentation in the program. Static code analysis does pass, however this is due to the max line length being set to 120, when the standard is 80 for this project.

The quantity and variety of tests is very good, with each application having tests for all possible cases, with mocks being used throughout the applications for the functions they rely on. The appfactory test could be better, since the test relys on the echo command, which goes against the unit test principle of a unit test being able to sandbox test the unit in question, without the need of any other functions or classes in the program. This would be better if a mock command was initialized from cmdApp. There is 100% coverage on all relevant files.

The use of parameterized expand greatly helps code readability, however would be best if implemented throughout the program, not just in test_evaluator. All other tests not mentioned are highly satisfactory.

### Error Handling Score: 3

All of the necessary errors were caught for the applications which handle file opening and reading.

As mentioned in the design section, the choice of using an exception string means that all errors thrown throughout the program will only have a string representation. If custom errors were made which could wrap these strings, it would be much easier to test where an error has been thrown from, and whether it’s a developer thrown error, or a user thrown error