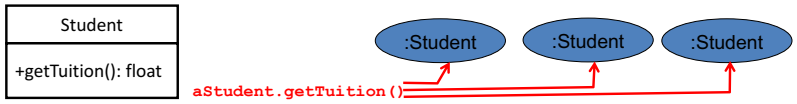
Visitor Design Pattern

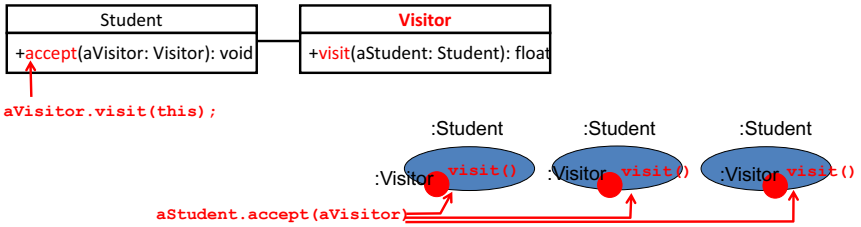
Visitor Design Pattern

- Intent
  - Separate (or decouple) a set of objects and the operations to be performed on those objects.

- In a traditional (or normal) design, if an operation is performed on some objects, it is defined as a method of a class(es) for those objects.

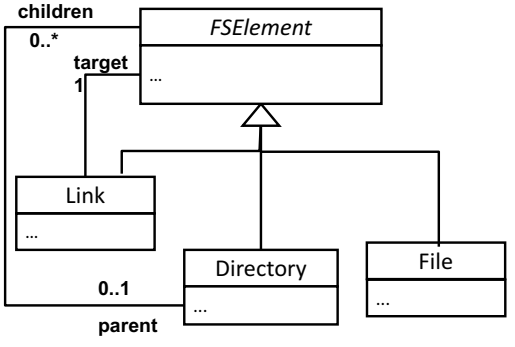


- With *Visitor*, the operation is defined in Visitor.

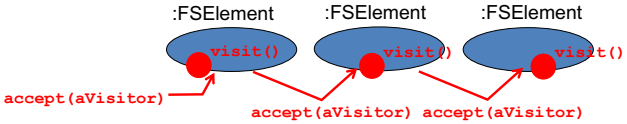


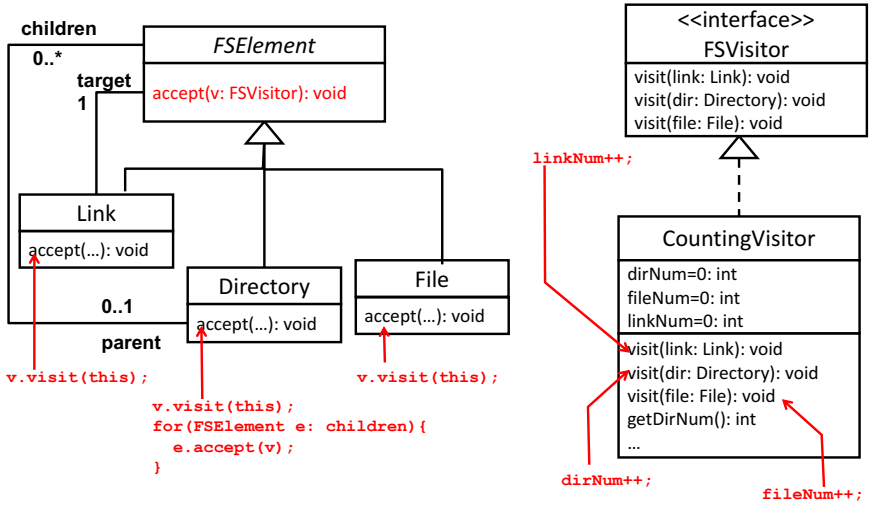
File System Examples (1/5)

- Count the number of directories, the number of files and the number links in a file system



- With *Visitor*, an operation to count FS elements is defined in a visitor.

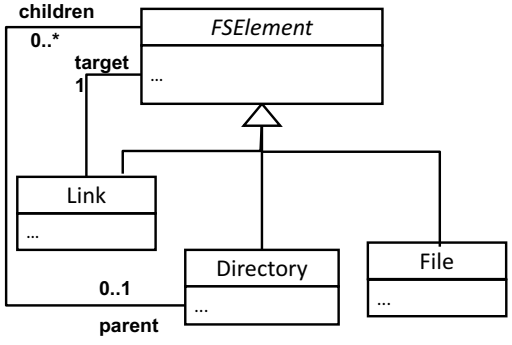


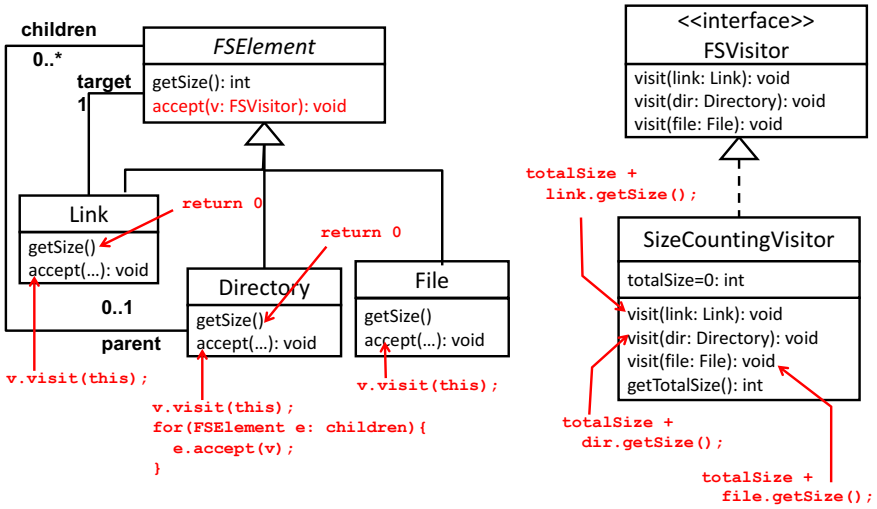



```
CountingVisitor visitor = new CountingVisitor();
rootDir.accept( visitor );
visitor.getDirNum(); visitor.getFileNum(); visitor.getLinkNum();
```


File System Examples (2/5)

- Count the total disk utilization in a file system

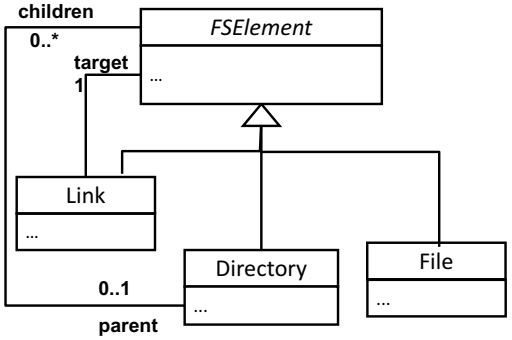


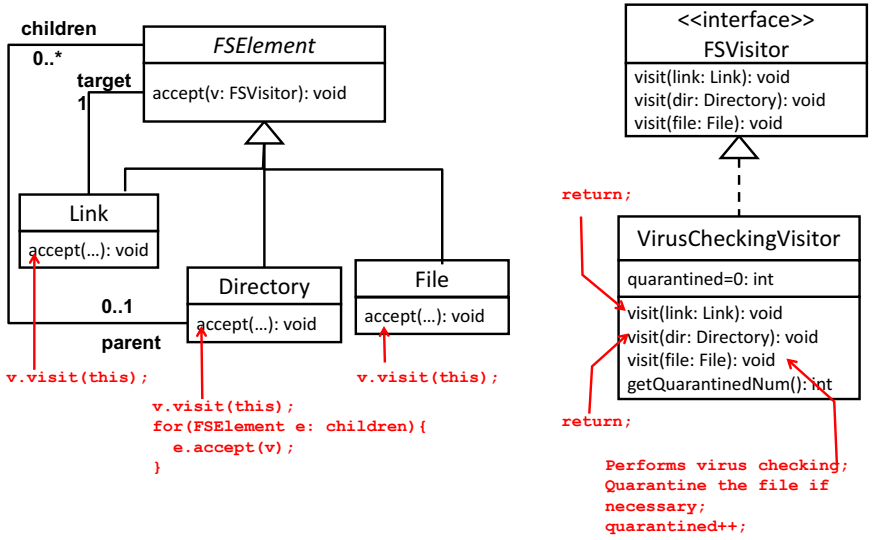



```
SizeCountingVisitor visitor = new SizeCountingVisitor();
rootDir.accept( visitor );
visitor.getTotalSize();
```

File System Examples (3/5)

- Perform virus check for each file in a file system

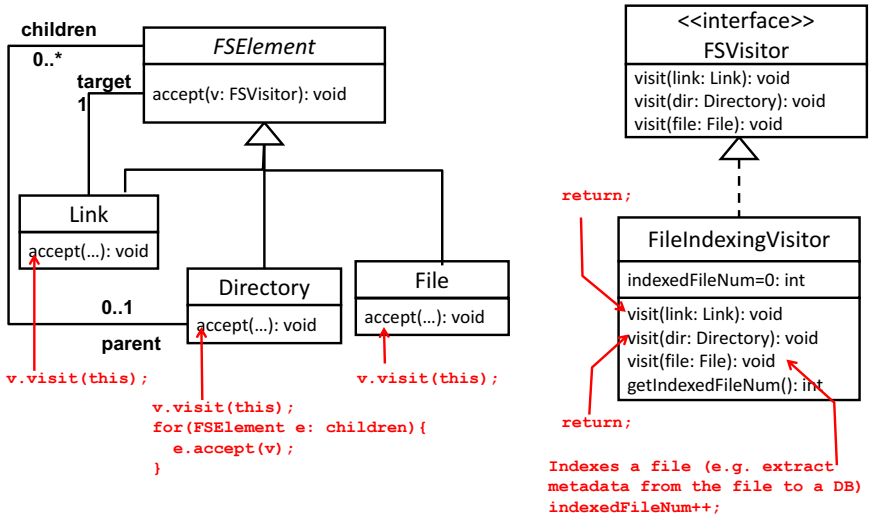





```
VirusCheckingVisitor visitor = new VirusCheckingVisitor();
rootDir.accept( visitor );
visitor.getQuarantinedNum();
```


File System Examples (4/5)

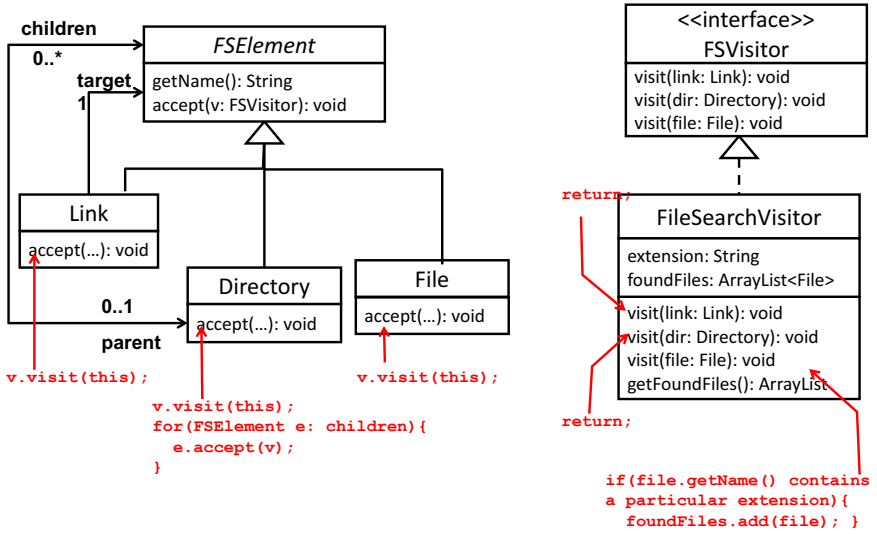
- Perform file indexing in a file system
  - Imagine an operating system's indexing service
    - e.g., Windows indexing service and Mac/iOS's Spotlight
  - Key functionalities
    - Crawl a file system to identify files
    - Index those files for later file searches.
      - Extract and keep each file's metadata
        - » Metadata: file's attributes (e.g., location, name, file type, author) and file's content




```
FileIndexingVisitor visitor = new FileIndexingVisitor();
rootDir.accept( visitor );
visitor.getIndexedFileNum();
```

File System Examples (5/5)

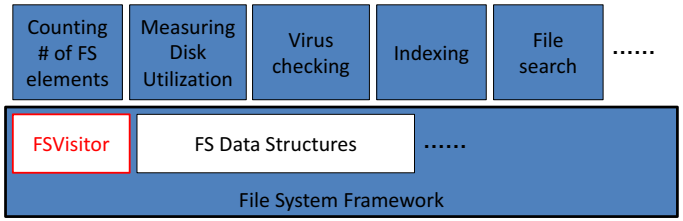
- Search files in a file system based on a given search criterion.
  - e.g. Search files that have a particular extension (e.g. .txt or .jpg).




```
FileSearchVisitor visitor = new FileSearchVisitor(".txt");
rootDir.accept( visitor );
visitor.getFoundFiles().size();
```


What's the Point?

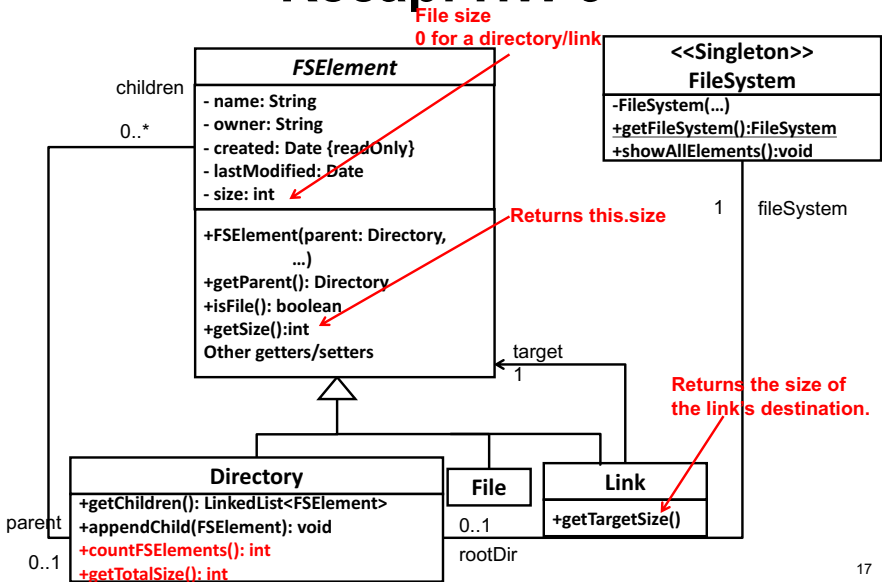
- Separating foundation data structures and the operations performed on those data structures.
  - It is easy to add, modify and remove operations.
  - Data structures can remain intact.



HW 10

- Implement FSVisitor and 3 visitor class.
  - CountingVisitor, SizeCountingVisitor and FileIndexingVisitor
  - FileIndexingVisitor does not have to implement actual indexing logic.
    - visit(file: File) can just print out a file's metadata (e.g. file name) on a shell.
- Due: the last day of the final's week.

Recap: HW 9



- Directory
  - `countFSElements()`: returns to total number of files and directories under a given directory.
  - `getTotalSize()`: returns the total disk consumption by all files and directories under a given directory.
- Keep these methods and compare them with CountingVisitor and SizeCountingVisitor
  - To understand the benefits of *Visitor*

Preliminaries: Road to Object-Oriented Design (OOD)

Brief History

- In good, old days… programs had no structures.
  - One dimensional code.
    - From the first line to the last line on a line-by-line basis.
    - "Go to" statements to control program flows.
      - Produced a lot of "spaghetti" code
        - » "Go to" statements considered harmful.
  - No notion of structures (or modularity)
    - Modularity: Making a chunk of code (module) self-contained and independent from the other code
      - Improve reusability and maintainability
        - » Higher reusability → higher productivity, less production costs
        - » Higher maintainability → higher productivity and quality, less maintenance costs

Modules in SD and OOD

- Modules in Structured Design (SD)
  - Structure = a set of variables (data fields)
  - Function = a block of code
- Modules in OOD
  - Class = a set of data fields and functions
  - Interface = a set of abstract functions
- Key design questions/challenges:
  - how to define modules
  - how to separate a module from others
  - how to let modules interact with each other

SD v.s. OOD

- OOD
  - Intends coarse-grained modularity
    - The size of each code chuck is often bigger.
  - Extensibility in mind in addition to reusability and maintainability
    - How easy (cost effective) to add and revise existing modules (classes and interfaces) to accommodate new/modified requirements.
    - How to make software more flexible/robust against changes in the future.
  - How to gain reusability, maintainability and extensibility?

How to Gain Reusability, Maintainability and Extensibility?

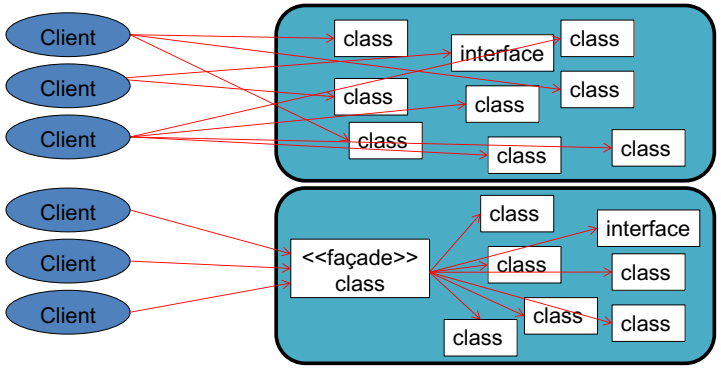
- Design patterns are identified and documented to answer this question.
  - You can learn how you can/should organize your code to gain these properties.
- Recall, for example,
  - *Command*
    - How to make commands extensible?
    - How to make command senders and command receivers maintainable?
  - *State*
    - How to make state-dependent behaviors (operations) maintainable?
  - *Visitor*
    - How to make visitors (i.e. operations to be applied on a set of data structures) extensible?
    - How to make the set of data structures maintainable?

- You can learn about reusability, maintainability and extensibility only through writing and running your own code.
  - Through DOING, not listening to someone, reading something or drawing mental pictures.

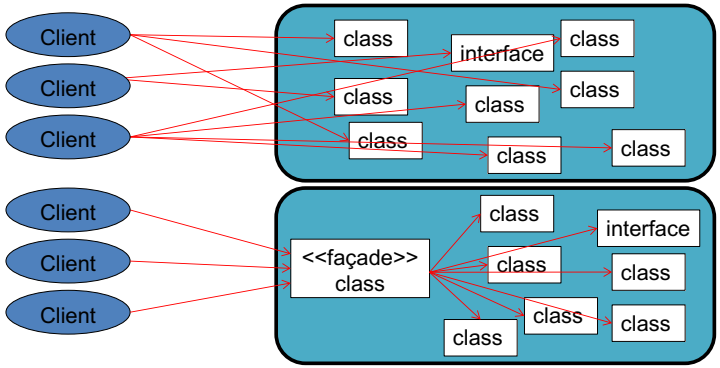
Façade Design Pattern

Façade

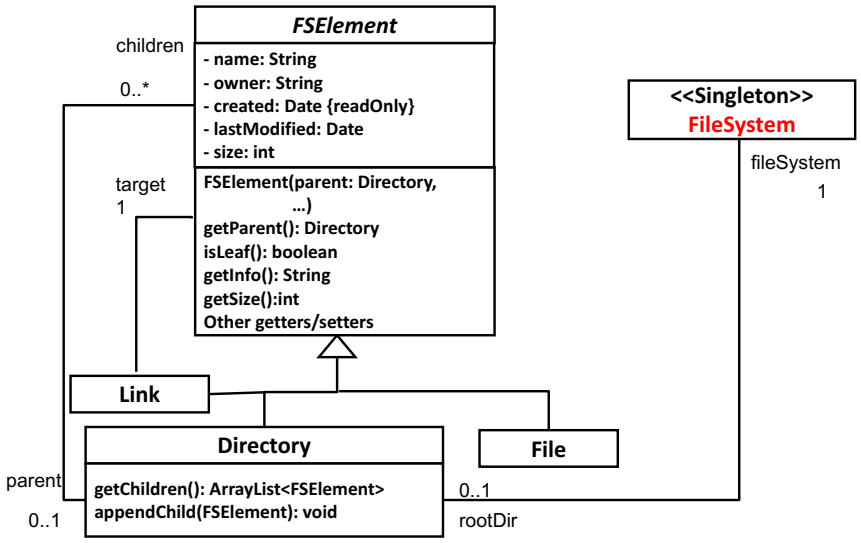
- Intent
  - Provide a unified and higher-level interface (or primary point of contact) to a set of data structures in a system.



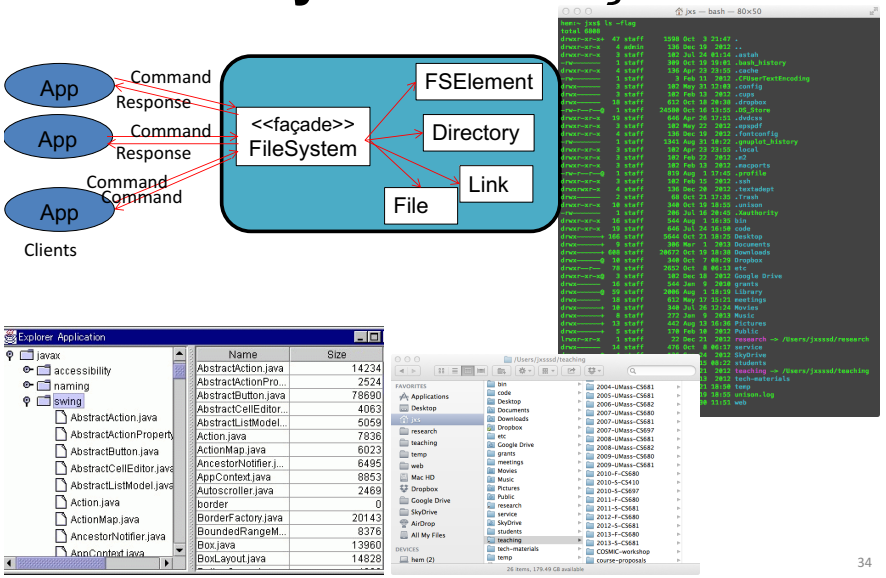
- Benefits
  - Makes those data structures easier to use for clients.
  - Decouple (or loosely couple) those data structures and clients



Recap: File System



FileSystem as *Façade*



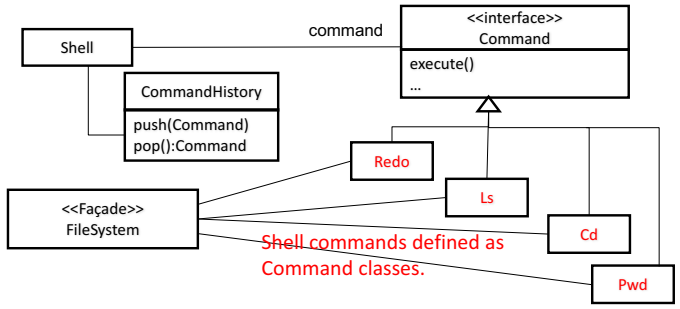
HW 11

- Implement a shell for your FS elements.
  - NOT a GUI shell, but a CUI (character UI) shell just like a Unix/Windows terminal.
- Implement individual shell commands with *Command.*
- Implement FileSystem as *Façade.*
- Implement a "pluggable" soring feature with Comparator (*Strategy*).

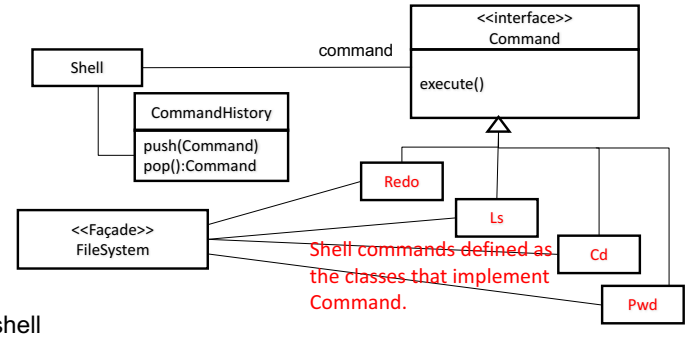
- Shell accepts the following commands:
  - pwd
    - Print the current working directory.
  - cd <dir name>
    - Change the current directory to the specified directory. Accept a relative (not absolute) directory name. Accept ".." (move to the parent directory of the current directory.)
  - cd
    - Change the current directory to the root directory.
  - ls
    - Print the name of every file, directory and link in the current directory.
  - dir
    - Print the information (i.e., kind, name, size and owner) of every file, directory and link in the current directory.
  - dir <dir/file name>
    - Print the specified directory's/file's information. Accept relative (not absolute) directory name. Accept ".."
  - mkdir <dir name>
    - Make the specified directory in the current directory.
  - rmdir <dir name>
    - Remove the specified directory in the current directory.
  - ln <target (real) dir/file> <link (alias) name>
    - Make a link
  - mv <dir/file> <destination dir>
    - Move a directory/file to the detonation directory
  - cp <dir/file> <destination dir>
    - Copy a directory/file to the destination directory
  - chown
    - Change the owner of a file/directory
  - history
    - Print a sequence of previously-executed commands.
  - redo
    - Redo the most recently-executed command.
  - sort
    - Sort directories and files in the current directory

Designing FS Commands with *Command*

- Why *Command*?
  - There exist several (potentially many) clients/apps for each command.
  - Each command has relevant arguments/options.
  - New commands will be added for sure in the future.
  - Existing commands may be modified/updated in the future.
  - Need to record/log command history.
    - "history" command, "redo" command

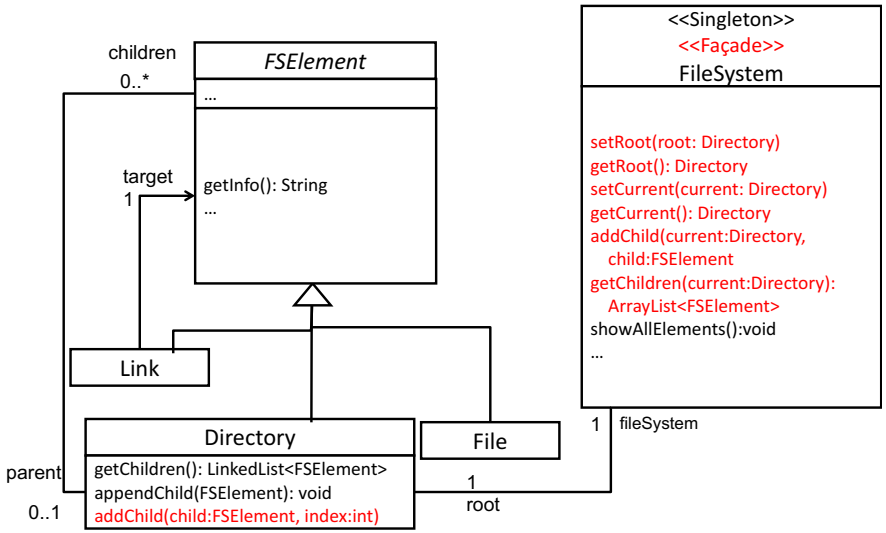


Designing FileSystem as *Façade*



- The shell
  - Receives a command (e.g. "cd" command) from the user
  - Creates an instance of a corresponding Command class (e.g. Cd)
  - Calls execute() on the instance
- execute() calls a method(s) of FileSystem to run the command on FS elements.
  - That is, FileSystem serves as Façade.

Example (not Complete) Methods in FileSystem



An Example Interaction among User, Shell and FileSystem

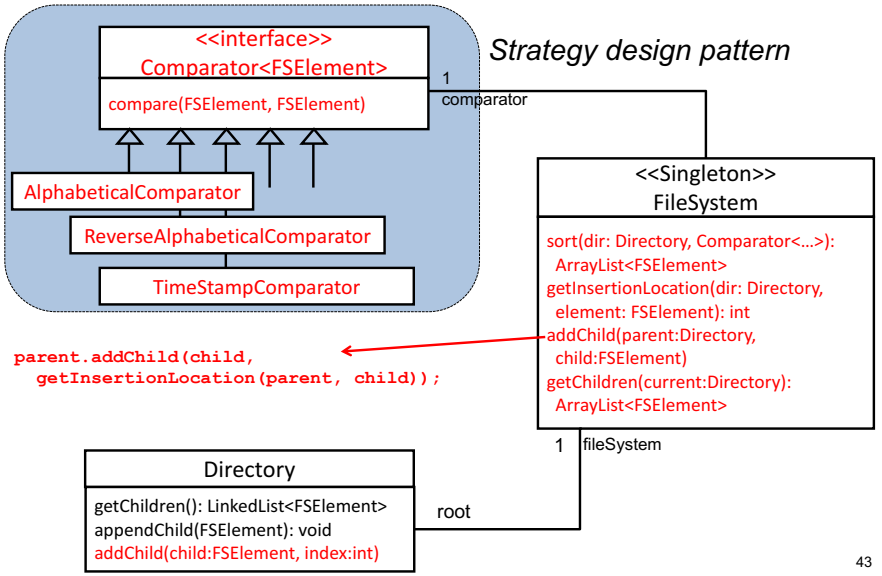
- The shell
  - prints out a prompt like ">",
  - lets the user enter a command and parses it,
    - Assume the user enters "cd …" as a command.
  - Creates an instance of Cd, and
  - Calls execute() on the Cd instance.
- execute()
  - implements the logic of a command by calling a method(s) of FileSystem, and
    - execute() of the Cd class
      - Checks if the destination directory exists by calling getChildren(), etc. and moves to the destination by calling setCurrent().
      - calls setCurrent(getRoot()) if "cd" command has no parameters.
  - returns any output message to Shell.

Sorting FS Elements

- Example soring policies
  - Alphabetical
  - Reverse alphabetical
  - Timestamp-based (e.g. "last-modified date"-based)
  - Element kind based (e.g. directories listed first followed by files and links, file type based)

- It is not a good idea to hardcode sorting logic in Directory.
  - Whenever a new sorting policy is required, you need to modify Directory.
- Better idea: Make Directory open-ended for various sorting policies (i.e., make Directory loosely-coupled from sorting policies)
  - Allow each FS user to select a sorting policy dynamically
  - Allow FS developers to add new sorting policies in a maintainable manner.
    - Have them add extra code (classes) rather than modify Directory.
- Solution: Use *Strategy* (Comparator).

Soring FS Elements with Comparator



  - addChild() always follows the default (alphabetical) sorting policy.
    - Directory always retains alphabetically-sorted FS elements.
    - getChildren() returns alphabetically-sorted elements.
  - sort(Directory, Comparator<FSElement>) re-sorts FS elements based on a custom (non-default) sorting policy, which is indicated by the second parameter, and returns re-sorted elements.
    - Directory does not have to retain the re-sorted elements.
    - Implement at least one custom sorting policy (e.g., timestamp-based)

- All previous HW solutions for file system development must be integrated into a single code base.
- Unit tests are required for all major public methods of all classes.