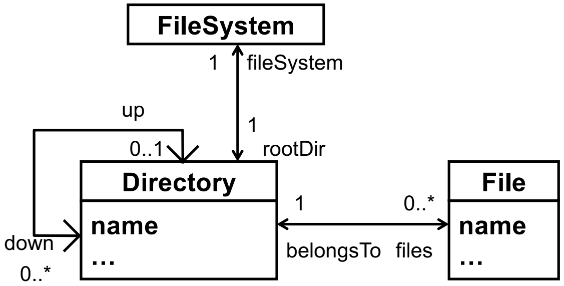
Composite Design Pattern

Composite Design Pattern

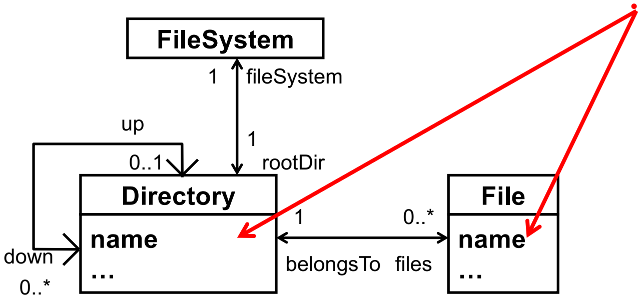
- Intent
  - Compose objects into a tree structure to represent a part-whole hierarchy.
  - Allow clients (of a tree) to treat individual objects and compositions of objects uniformly.

A Design Exercise: File System

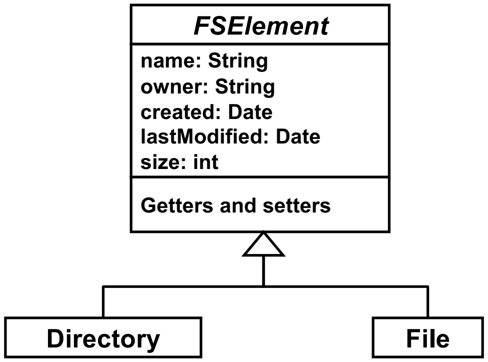
- A file system consists of directories and files.
- Each file exists in a particular directory.
- Each directory can contain multiple files.
- Directories form a tree structure.
  - Every directory has its parent directory, except the root directory.
  - Each directory can have multiple sub directories.
- Each directory and file has the following properties:
  - Name, owner's name, date of creation, date of the last modification and disk utilization (i.e., file/directory size)



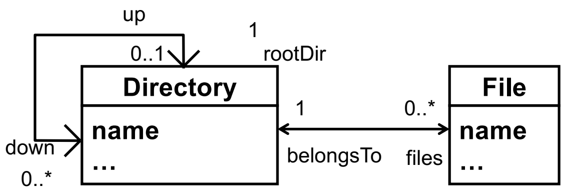
- A file system consists of directories and files.
- Each file exists in a particular directory.
- Each directory can contain multiple files.
- Directories form a tree structure.
  - Every directory has its parent directory, except the root directory.
  - Each directory can have multiple sub directories.
- Each directory and file has the following properties:
  - Name, owner's name, date of creation, date of the last modification and disk utilization (i.e., file/directory size)



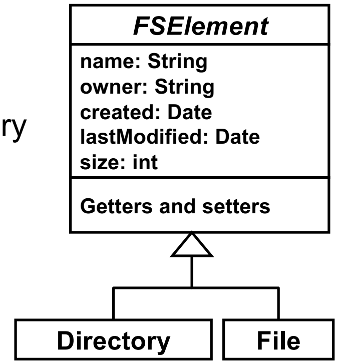
- Each directory and file has the following properties:
  - Name, owner's name, date of creation, date of the last modification and disk utilization (i.e., file/directory size)



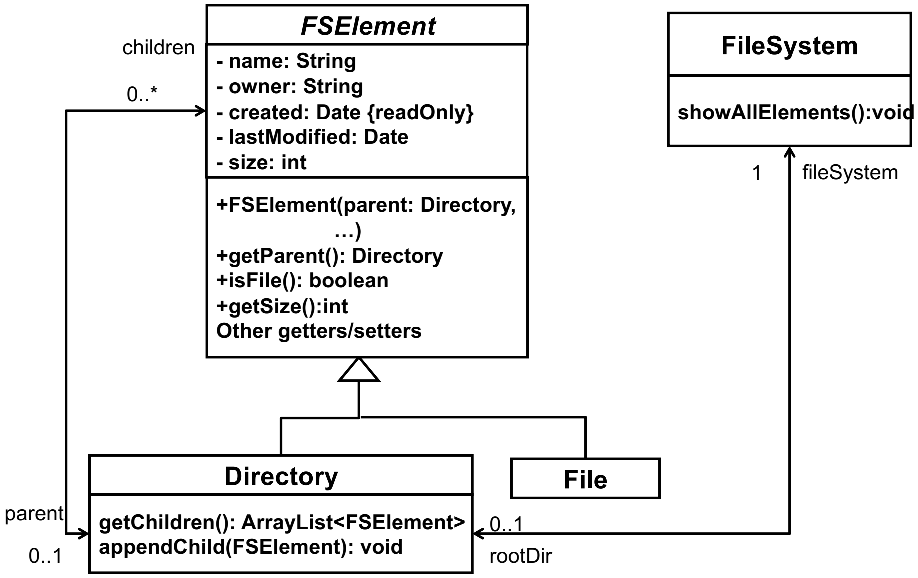
- A directory is never transformed to be a file.
- A file is never transformed to be a directory.



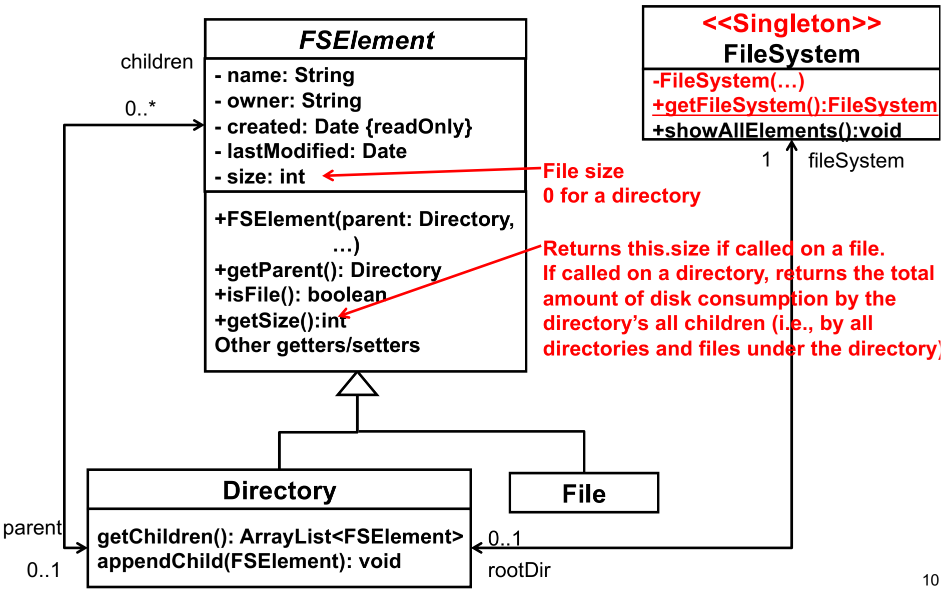
- How can we design directory-to-directory structures?
- How can we design file-to-directory structures?



Using *Composite*…



HW12: Implement this.



children

0..*

FSElement

- name: String
- owner: String
- created: Date {readOnly}
- lastModified: Date
- size: int

+FSElement(parent: Directory, …)
+getParent(): Directory
+isFile(): boolean
+getSize():int
Other getters/setters

File size
0 for a directory

Returns this.size if called on a file.
If called on a directory, returns the total amount of disk consumption by the directory's all children (i.e., by all directories and files under the directory)

<<Singleton>>
FileSystem

-FileSystem(…)
+getFileSystem():FileSystem
+showAllElements():void

1

fileSystem

Directory

getChildren(): ArrayList<FSElement>
appendChild(FSElement): void

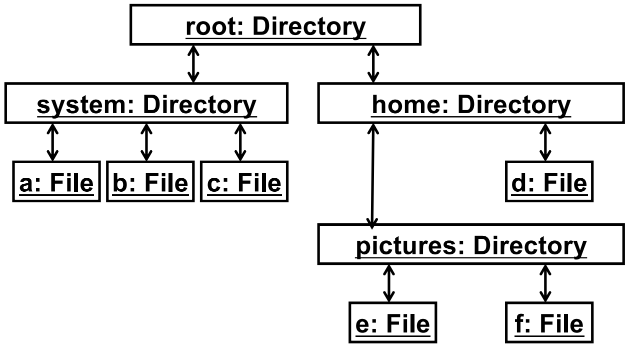
File

parent

0..1

0..1

rootDir



root: Directory

system: Directory

home: Directory

a: File

b: File

c: File

d: File

pictures: Directory

e: File

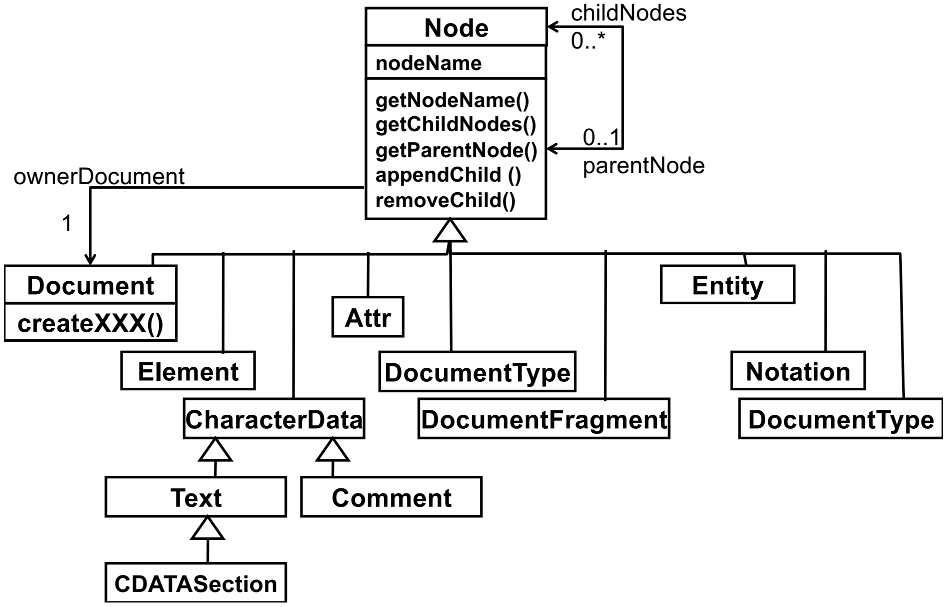
f: File

- Make this tree structure in your test case.
  - Assign values to data fields (size, owner, etc.) as you want.
  - Call getSize() on the root directory.
  - Call showAllElements() to print out this tree structure.
    - You can define your own textual format.

Another Example: Document Object Model (DOM)

- Document Object Model (DOM)
  - A parser interface for XML parsers.
  - Specification: www.w3c.org
    - Level 1, DOM Core
  - Implementations:
    - Has been implemented by many libraries/frameworks.
    - Has been implemented by virtually all major languages.
    - e.g., Java API (javax.xml)

Document Object Model (DOM)



Node

nodeName

getNodeName()
getChildNodes()
getParentNode()
appendChild ()
removeChild()

childNodes

0..*

0..1

parentNode

ownerDocument

1

Document

createXXX()

Element

CharacterData

Attr

DocumentType

DocumentFragment

Entity

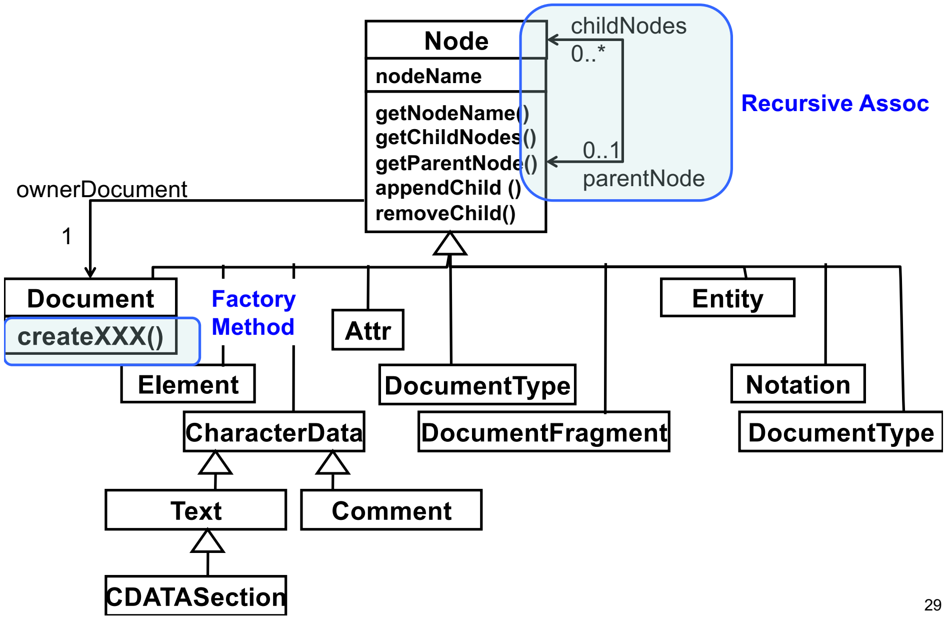
Notation

DocumentType

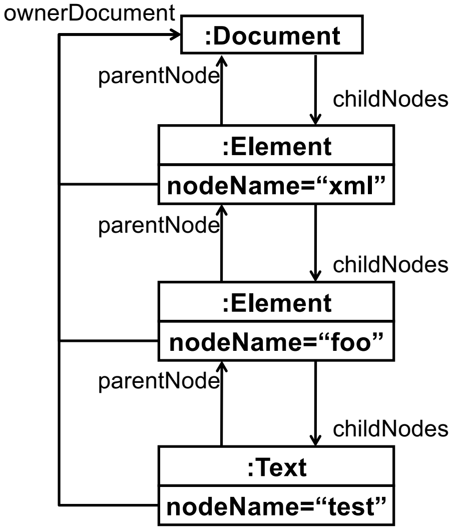
Text

Comment

CDATASection



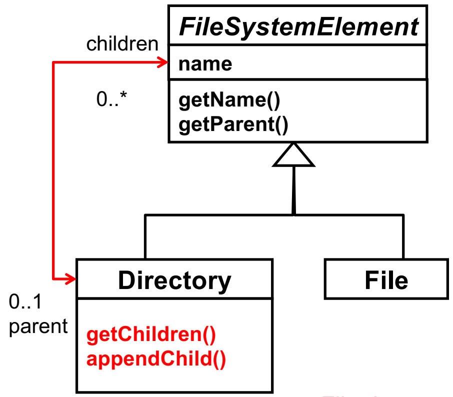
Example Instances of DOM Classes



```
<xml>
  <foo>test</foo>
</xml>
```

Two Variants of *Composite*

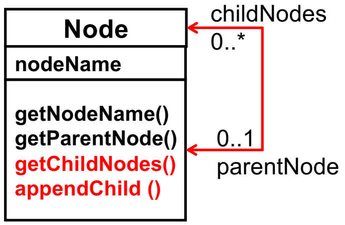
Variant #1



Directory encapsulates child-handling methods.

File does not have child-handling methods.

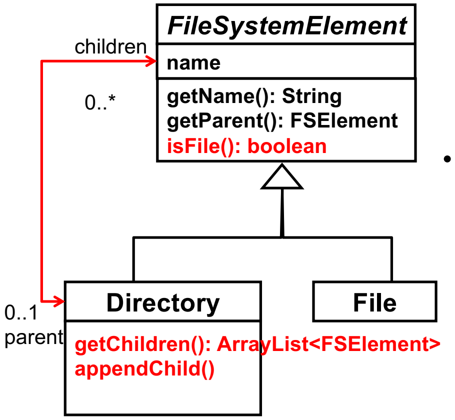
Variant #2



Node can represent directories and files.

Both directories and files have child-handling methods.

Variant #1



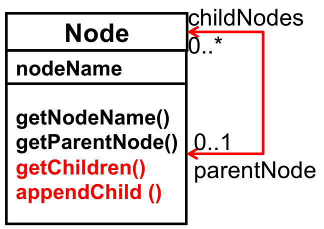
Directory encapsulates child-handling methods.

File does not have child-handling methods.

- Pros
  - No need to implement unnecessary (i.e., child-handling) methods in File.
  - Type safe
- Cons
  - User/client code has to be aware of the type of an FSElement (Directory or File).
    - Need downcasting.

  -
    ```
    Iterator itr = aDir.getChildren().iterator();
    while( itr.hasNext() ){
      FSElement elem = itr.next();
      if( !elem.isFile() ){
        ((Directory) elem).getChildren(); }
      else{
        … } }
    ```
  - User/client code has to be modified when new subclasses are added.

Variant #2



Node

nodeName

getNodeName()
getParentNode()
getChildren()
appendChild ()

childNodes 0..*

0..1 parentNode

Node can represent directories and files.

Both directories and files have child-handling methods.

- Pros
  - User/client code does not have to know the type of a file system element.
    - No need to do downcasting.

  -
    ```
    Iterator itr = aNode.getChildren().iterator();
    while( itr.hasNext() ){
        Node elem = itr.next();
        elem.getName();
        elem.getChildren(); } // No if statement here.
    ```
  - User/client code can be intact even if new types of nodes are added.
  - Less number of classes
- Cons
  - Need to implement child-handling methods in a relatively ugly way.
    - Generate an error.
    - Throw an exception.
  - Lower modularity and lower type safety
  - Many methods/variables in a single class

Which Variant to Use?

- How often do you expect to change the structure of classes?
  - Adding and removing subclasses of FSElement?
  - Often
    - Variant #2 would make more sense.
      - User/client code can be independent from the changes in subclasses.
  - Rare
    - Variant #1 would make sense too.
      - More type safe.
      - Less error handling.

A Design Decision/Rationale in DOM

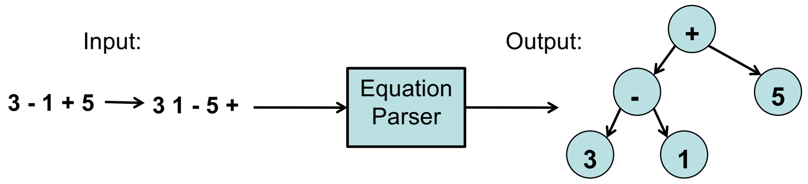
- Class structure may often change as the DOM specification evolves.
  - The structure of XML documents can change/evolve independently from DOM’s API design.
    - due to future updates in the XML specification.
  - Backward compatibility is important for user/client code.
- DOM designers chose variant #2 .

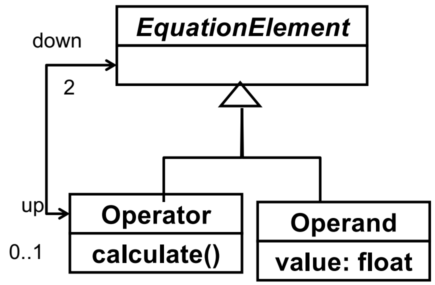
A Design Decision/Rationale in FS

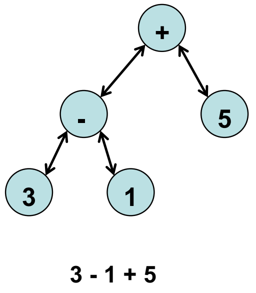
- The variety of subclasses is very limited.
  - Changes are rare on those subclasses.
- Variant #1 makes more sense.

An Example of *Composite*

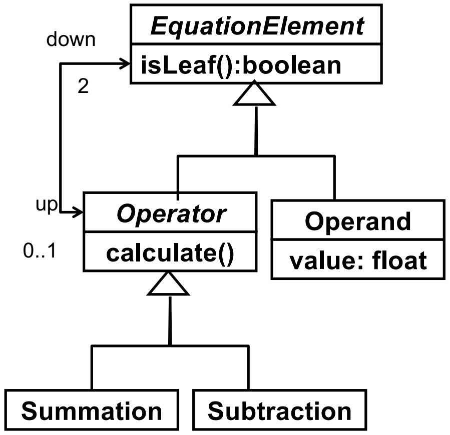
- Assume you are coding a parser to parse equations in a textual form.
  - Input: textual/string representation of an equation
  - Output: equivalent in-memory representation
    - Tree structure
      - its leaf nodes represent operands
      - the root and intermediate nodes represent operators.





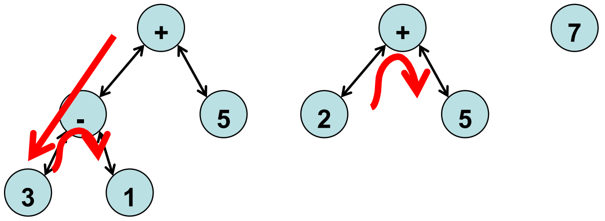


- With the Composite design pattern, design the data structures (i.e., operators and operands), as a class diagram, to build this in-memory tree representation
  - Consider summation and subtraction operators only
    - No multiplication, division and other operators
  - Consider binary operators only, but assume using multiple binary operators in an equation (e.g., 3 - 1 - 5)
  - No need to consider precedence rules and parentheses



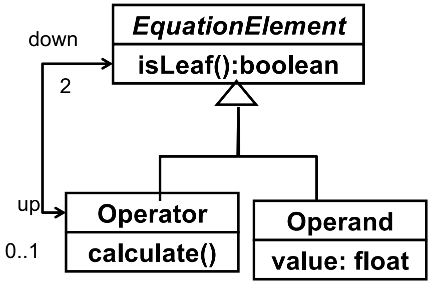
- Conditional statements can be eliminated in calculate()

- This parser requires a depth-first traversal policy.
  - Starts with the “deepest” and “left-most” leaf node
  - Traverse all nodes in the same layer
  - Goes up to a higher layer



Design Decision/Rationale in an Equation Parser

- Subclasses are only Operand and Operator.
- Changes are rare in the structure of equations.
- Variant #1 makes more sense


down
2
EquationElement
isLeaf():boolean
up
0..1
Operator
calculate()
Operand
value: float