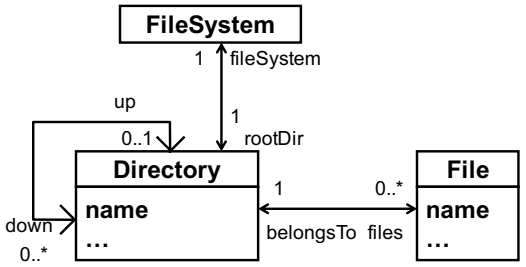
Composite Design Pattern

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.

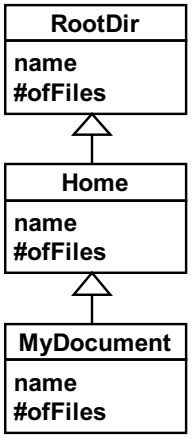
A Design Exercise: File System

- 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)

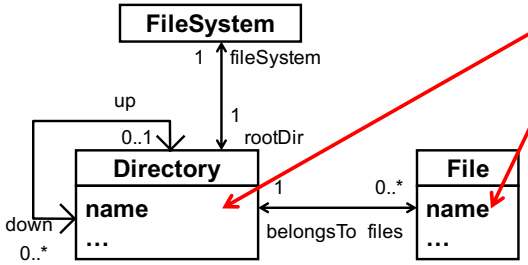


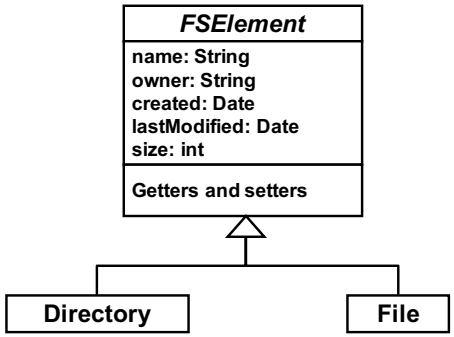
- A file system consists of directories and files.
- Each file exists in a particular directory.
- Each directory can contain multiple files.
- Directories form a tree structure.
  - Every directory has its parent directory, except the root directory.
  - Each directory can have multiple sub directories.
- 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)

Don't do this.

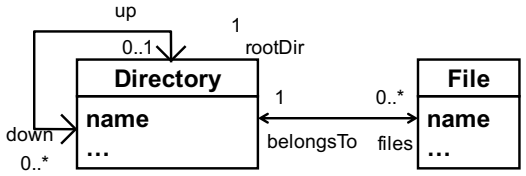


- Each directory and file has the following properties:
  - Name, owner's name, date of creation, date of the last modification and disk utilization (i.e., file/directory size)

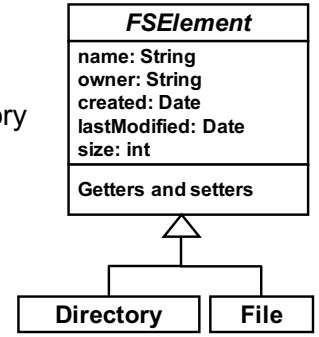




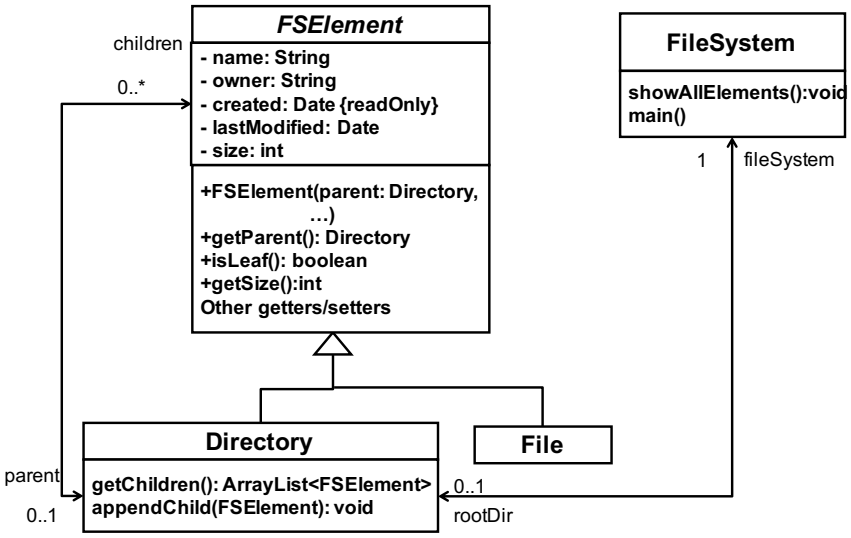
- A directory is never transformed to be a file.
- A file is never transformed to be a directory.



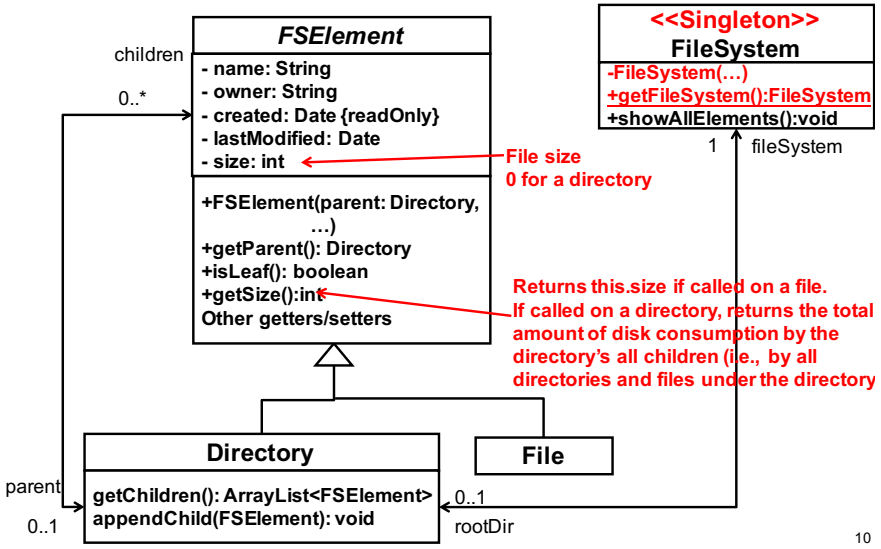
- How can we design directory-to-directory structures?
- How can we design file-to-directory structures?

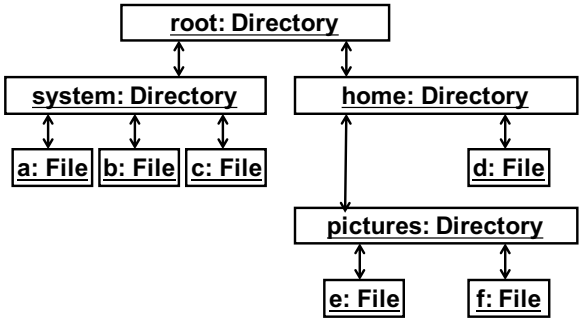


Using *Composite*…



HW14: Implement this.





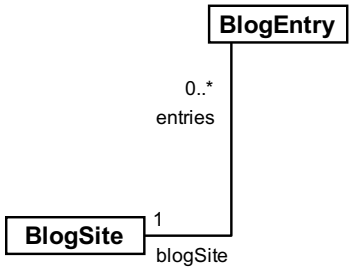
- Make this tree structure in your test case.
  - Assign values to data fields (size, owner, etc.) as you want.
  - Call getSize() on the root directory.
  - Call showAllElements() to print out this tree structure.
    - You can define your own textual format.

Another Exercise: Blog Site Structure

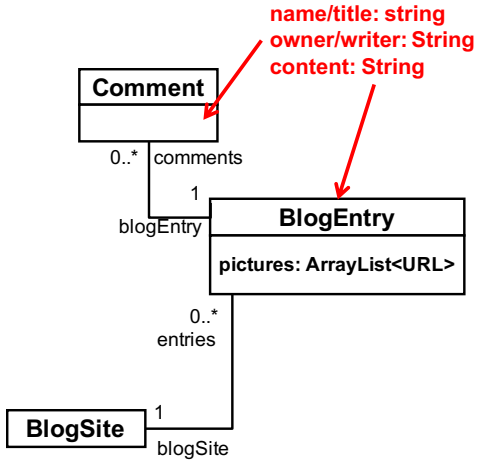
- A blog site is a collection of blog entries.
- Each blog entry has its name/title and owner's name.
- Each blog entry consists of
  - a textual content
  - zero or more pictures
    - Each picture is referenced by a URL.
  - zero or more comments.
    - Each comment has its own title, a textual comment body and its writer's name.
- Each blog entry and comment has the date of "last modified" and has a status:
  - public or private (draft/removed).

The First Step

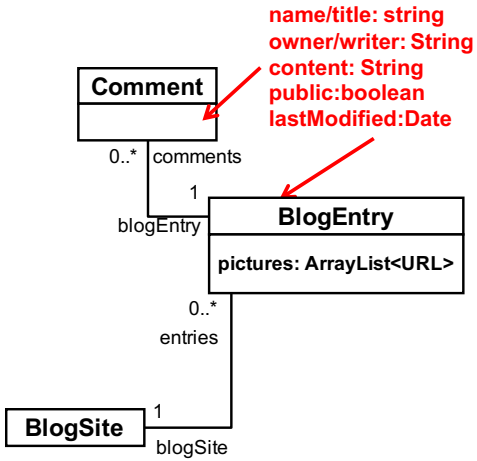
- A blog site is a collection of blog entries.

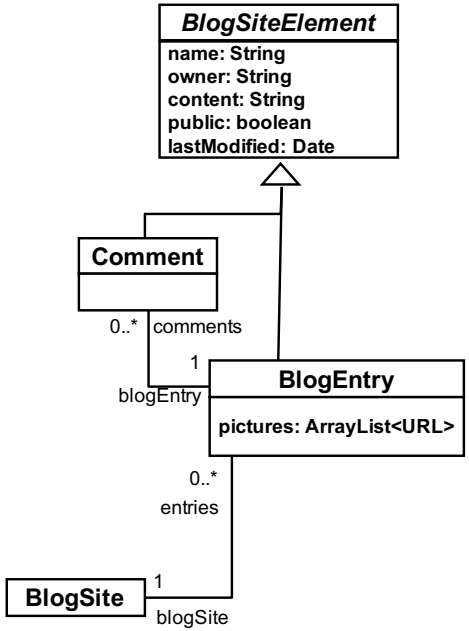


- A blog site is a collection of blog entries.
- Each blog entry has its name/title and owner's name.
- Each blog entry consists of
  - a textual content
  - zero or more pictures
    - Each picture is referenced by a URL.
  - zero or more comments.
    - Each comment has its own title, a textual comment body and its writer's name.

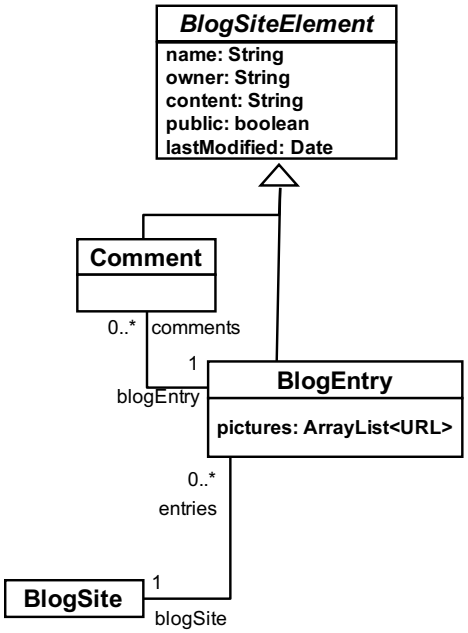


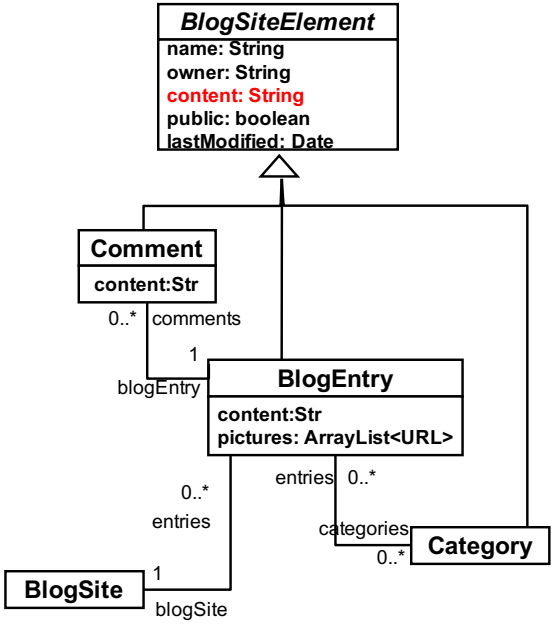
- Each blog entry and comment has the date of "last modified" and has a status:
  - Public v.s. private (draft/removed).





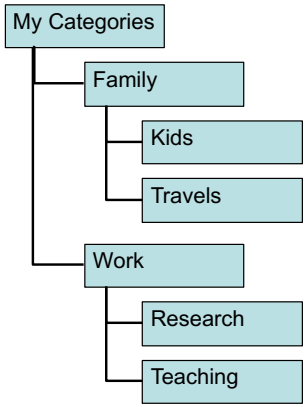
- Each blog entry can belong to one or more categories.
- Each category has its name, owner/creator's name, public/private status and "last modified" date.

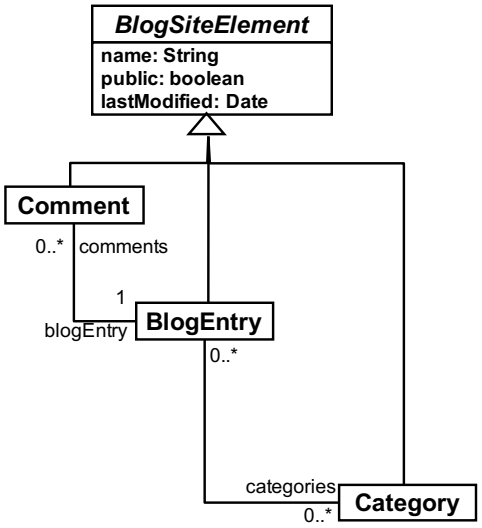




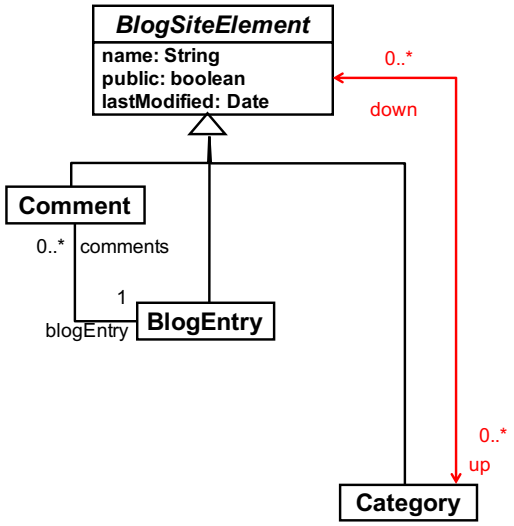
An Extension to Blog Structure

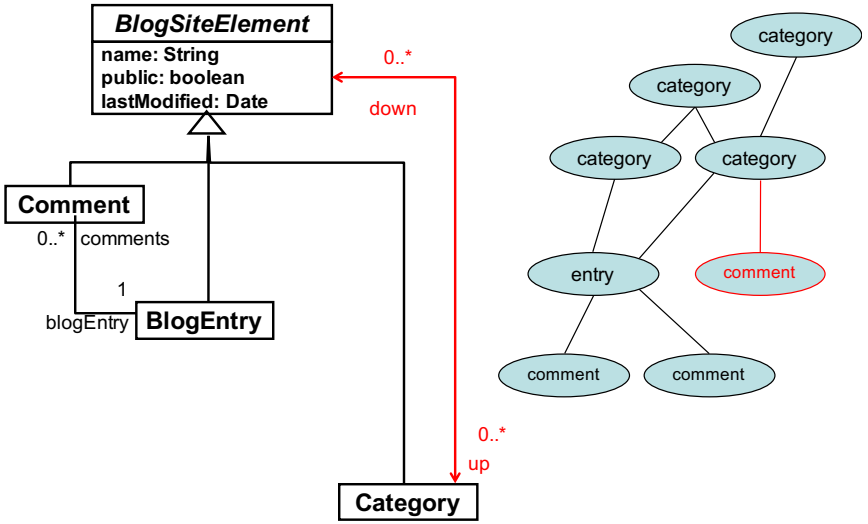
- Assume categories are hierarchical.
- What extension is required to accommodate this requirement?



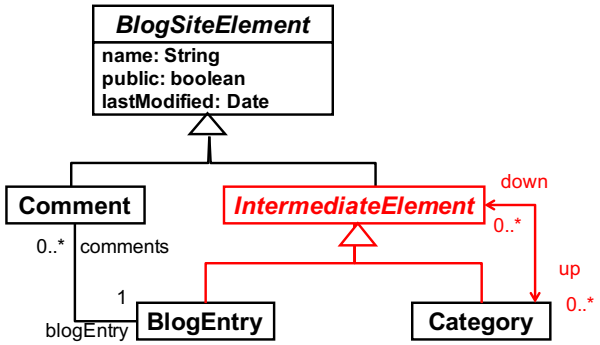


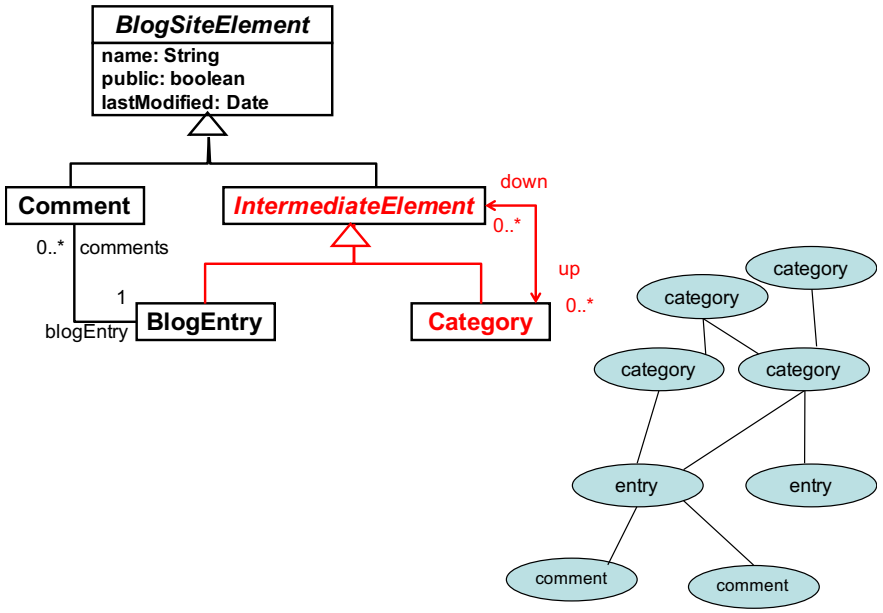
Design #1



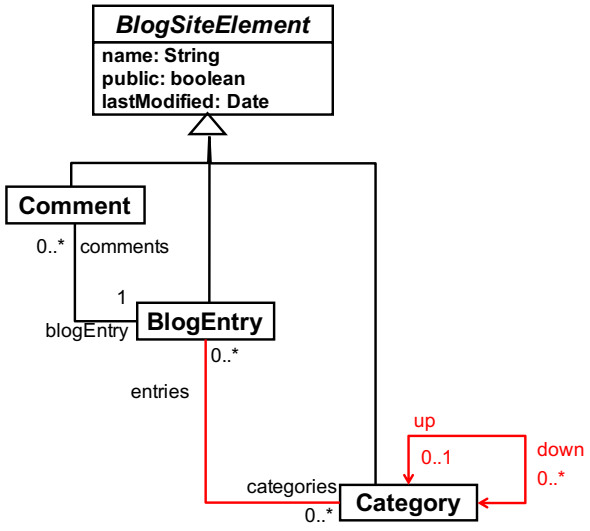


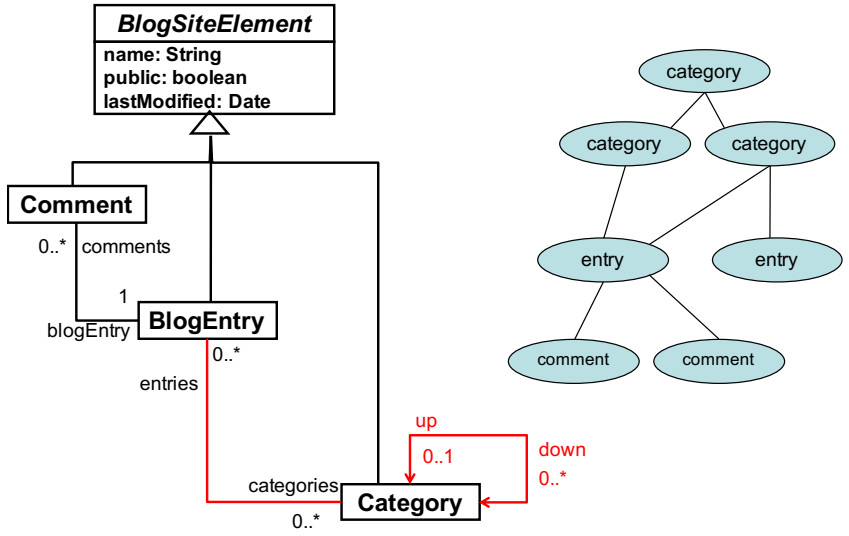
Design #2





Design #3

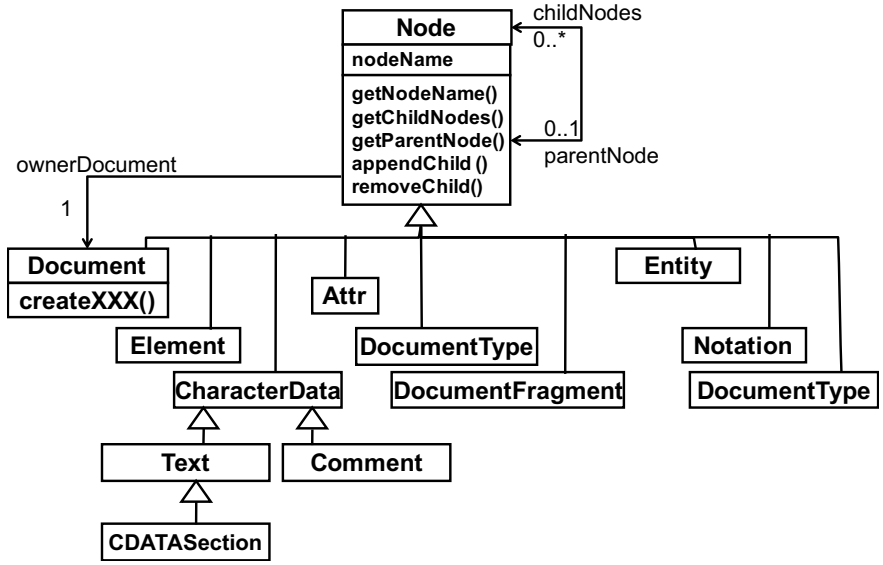


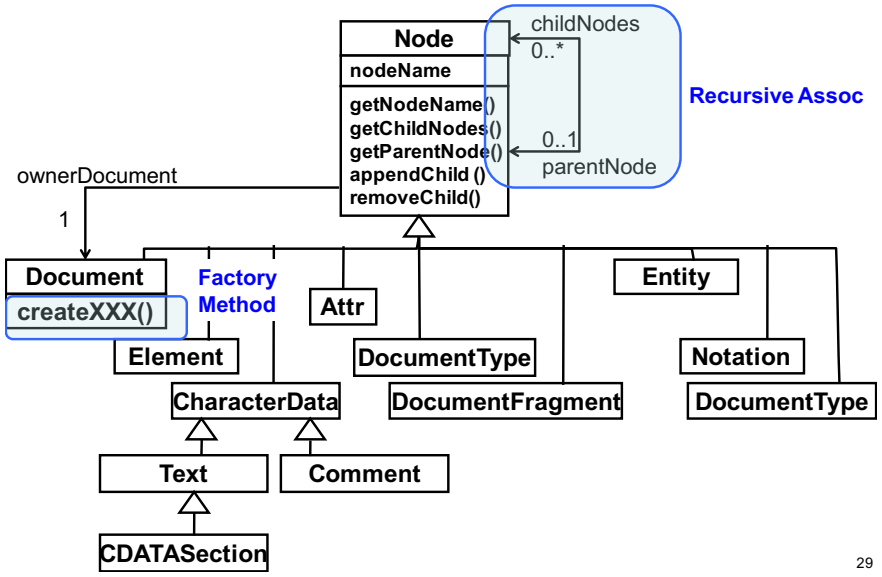


Another Example: Document Object Model (DOM)

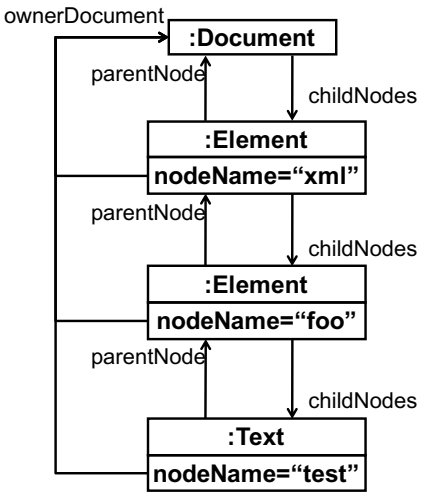
- Document Object Model (DOM)
  - A parser interface for XML parsers.
  - Specification: www.w3c.org
    - Level 1, DOM Core
  - Implementations:
    - Has been implemented by many libraries/frameworks.
    - Has been implemented by virtually all popular languages.
    - e.g., Java API (javax.xml)

Document Object Model (DOM)





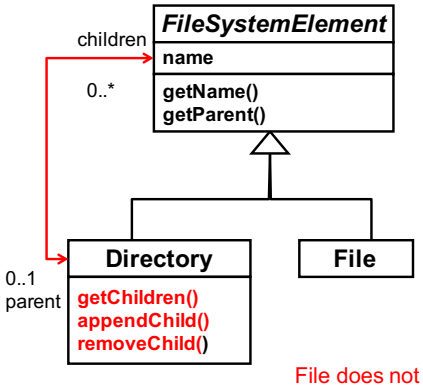
Example Instances of DOM Classes




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


Two Variants of *Composite*

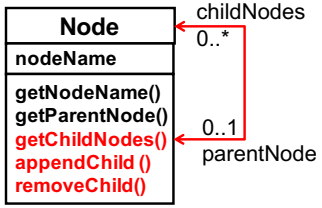
Variant #1



Directory encapsulates child-handling methods.

File does not have child-handling methods.

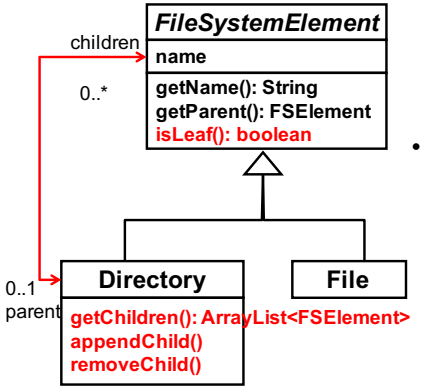
Variant #2



Node can represent directories and files.

Both directories and files have child-handling methods.

Variant #1

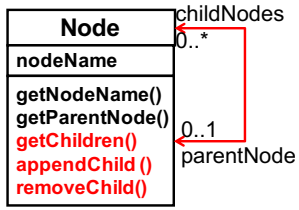


Directory encapsulates child-handling methods.

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

Variant #2



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.
    - Empty method body
    - Generate an error message.
    - Throw an exception.
  - Lower modularity and 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/removing/modifying subclasses?
  - 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.