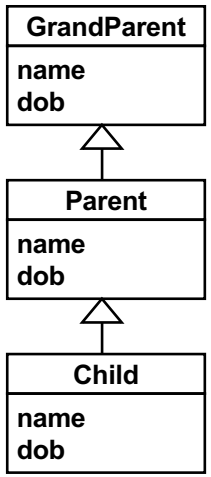
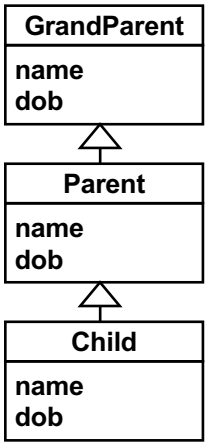
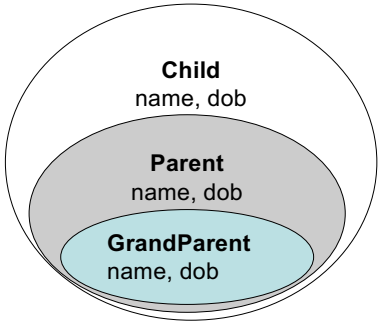
Implementing a Hierarchy of Objects with a Recursive Association

Wrong Design



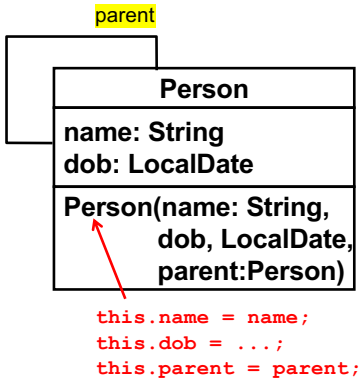
- A classical design error in/since mid ’80.
  - A parent “inherits” a grand parent’s data fields and methods.
  - A child “inherits” a parent’s data fields and methods.
- Found in an OOP textbook published in 2009. It’s still on the market.

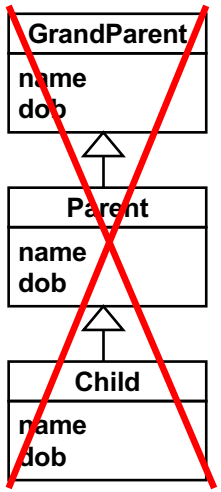




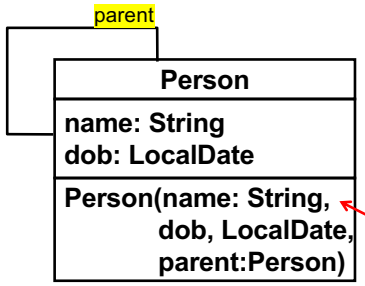
- A parent has two “name” data fields and two “dob” data fields!
- A child has three “name” data fields and three “dob” data fields!!

Right Design



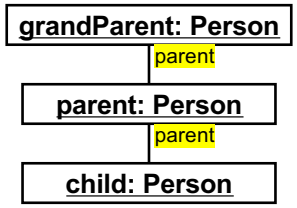


- Use a recursive association rather than class inheritance



Class diagram

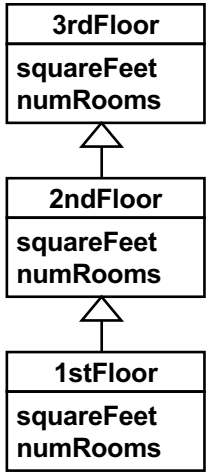
```
this.name = name;
this.dob = dob;
this.parent = parent;
```

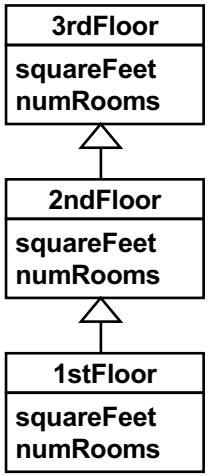


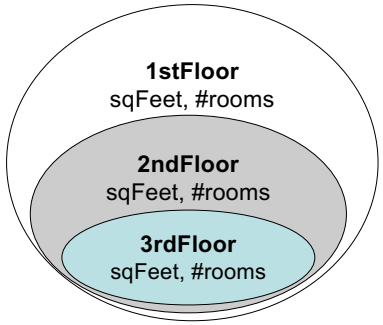
Class instances

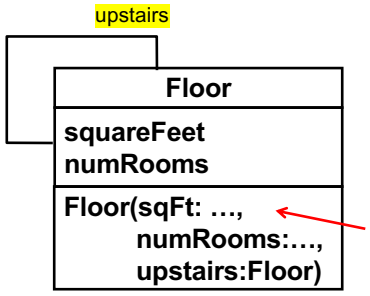
```
Person grandParent = new Person("grandpa", ..., null);
Person parent      = new Person("dad", ..., grandParent);
Person child       = new Person("me", ..., parent);

parent.getParent();
child.getParent();
```


Another Example



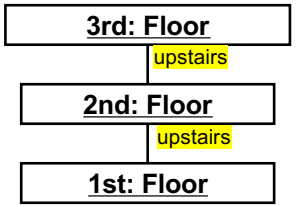






Class diagram

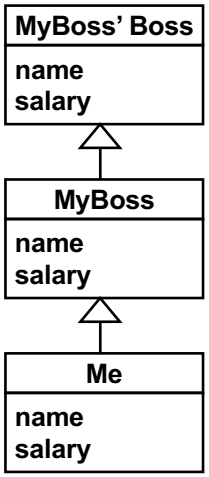
```
this.squareFeet = sqFt;
this.numRooms = numRooms;
this.upstairs = upstairs;
```



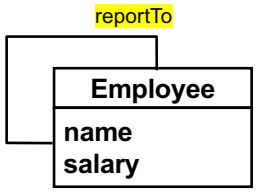
Class instances

```
Floor 3rd = new Floor(500, 2, null);
Floor 2nd = new Floor(700, 3, 3rd);
Floor 1st = new Floor(1000, 5, 2nd);

2nd.getUpstairs();
1st.getUpstairs();
```

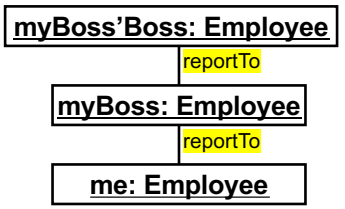
One More Example



Class diagram



Class instances




```
Employee mrBig = new Employee("bossboss", 100, null);
Employee boss= new Employee("boss", 70, mrBig);
Employee me= new Employee("me", 50, boss);

me.reportTo();
boss.reportTo();
```

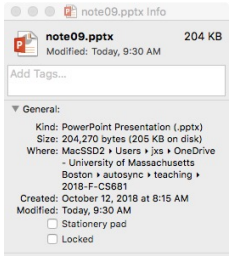

Composite Design Pattern

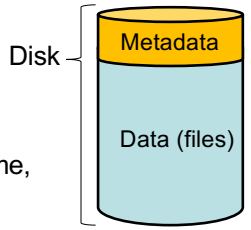
Composite Design Pattern

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

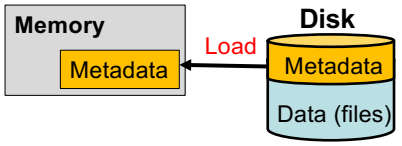
Example: Metadata Mgt in File Systems

- File system
  - Stores data as files in a structured way
  - Retrieves files
- Data to be stored
  - Data itself (file content)
  - Metadata (data about the data/file)
    - File name
    - File type (kind)
    - Physical file location in a disk
    - Logical file location (file path)
    - File size
    - File owner
    - File creation time, last-modified time (the time that the file was last modified), last-backed-up time, last-opened time
    - Access permission (who can see/read/edit a file)

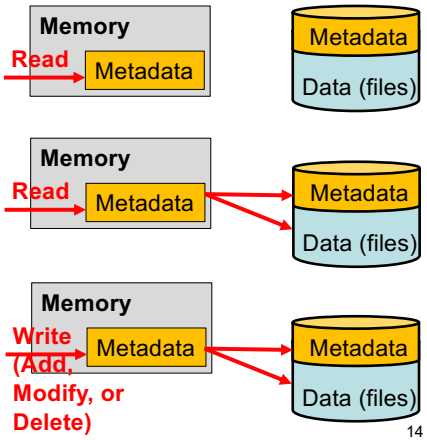




- An OS loads metadata to the main memory during its boot process.

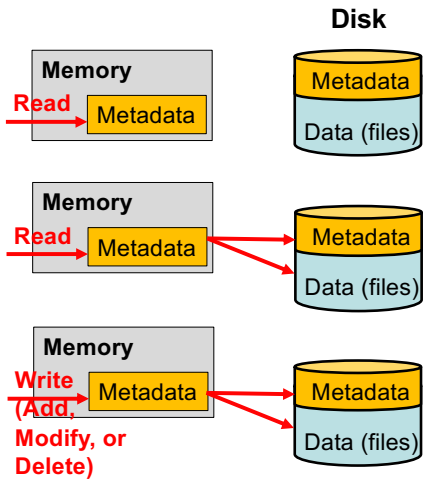


- Applications access (read and write) files through their metadata.
  - Read a file's metadata
  - Read a file's content
  - Add/store a new file
  - Modify a file's metadata and/or content.
  - Delete a file.



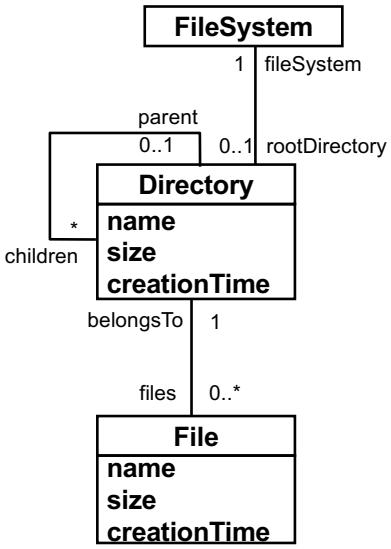
Metadata in a File System

- In-memory representation of a file system
  - Includes metadata of files and directories
  - Includes directory-to-directory structure
  - Includes file-to-directory structure
  - Gets updated as the file system is updated.



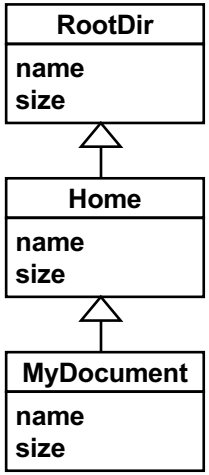
Requirements for Metadata Design

- 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 (i.e., file/directory name)
  - Size (i.e., file/directory size)
  - Creation timestamp (i.e., the time that a file/directory was created)



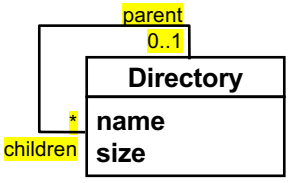
- A file system consists of directories and files.
- Each file exists in a particular directory.
- Each directory can contain multiple files.
- Directories form a tree structure.
  - Every directory has its parent directory, except the root directory.
  - Each directory can have multiple sub directories.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

Don't Do This.

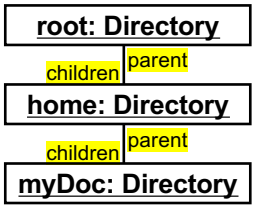


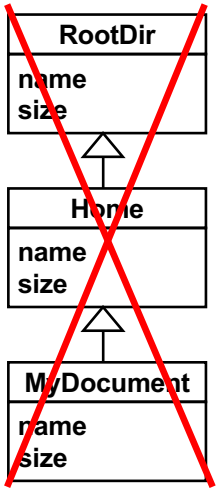
Do This.

Class diagram

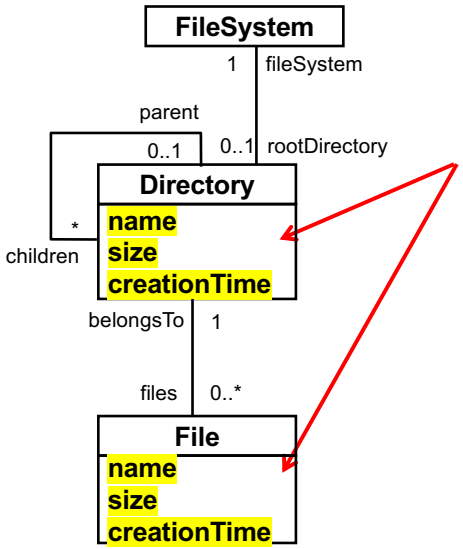


Object diagram






```
Directory root =  new Directory(null, ...);
Directory home =  new Directory(root, ...);
Directory myDoc = new Directory(home, ...);

root.getChildren();
home.getParent();
```

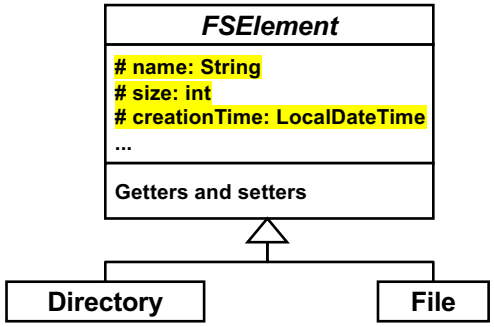
Directory and File have duplicated data fields and duplicated getter/setter methods.

Name, size, creation timestamp

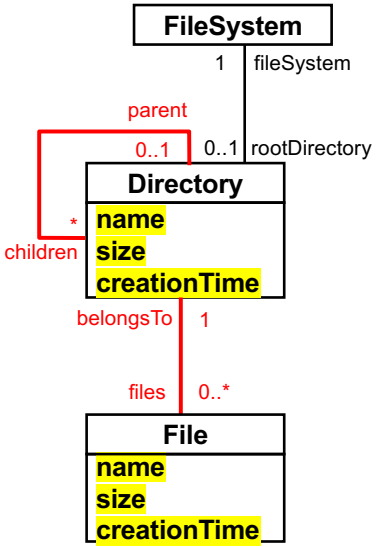
When a change is made for one of them, you need to change both classes.

When a new data field is added, you need to add it to both classes.

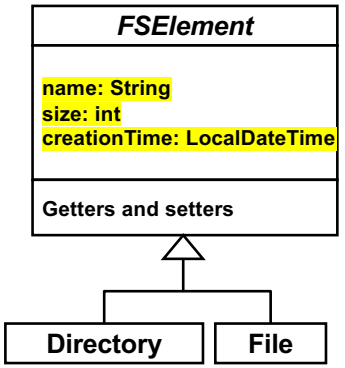
When a new type of file system elements is introduced (e.g., symbolic links, hidden system files, special directories such as *.app), you need to copy all existing data fields to a new class manually.



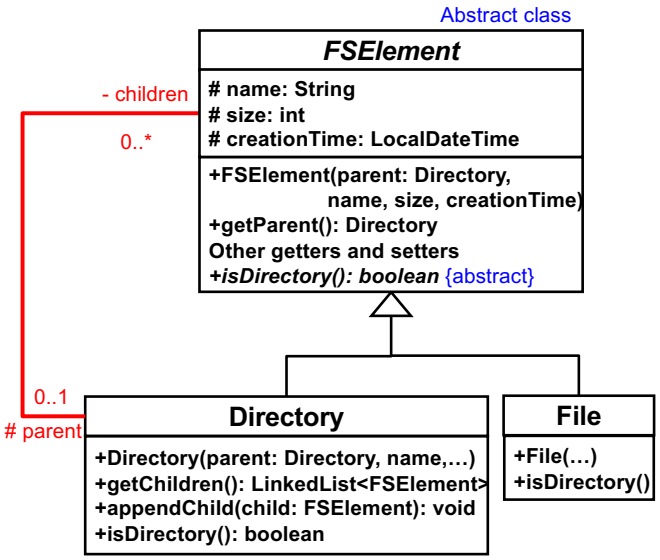
- Can eliminate the duplication (and potential laborious and error-prone maintenance work) by
  - moving those common data fields and their getter/setter methods to a super class of Directory and File.
- No problem to use class inheritance here
  - A directory is never transformed to be a file, and vice versa.

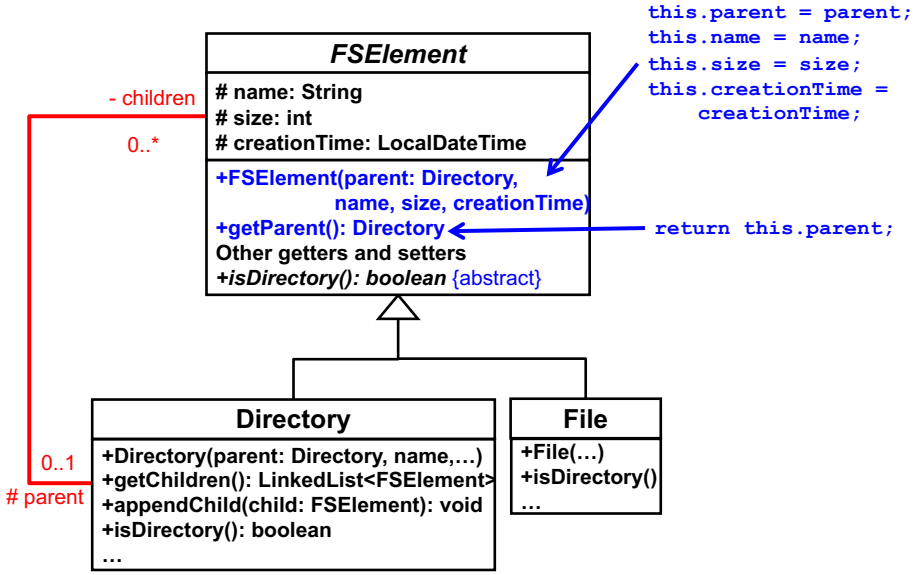


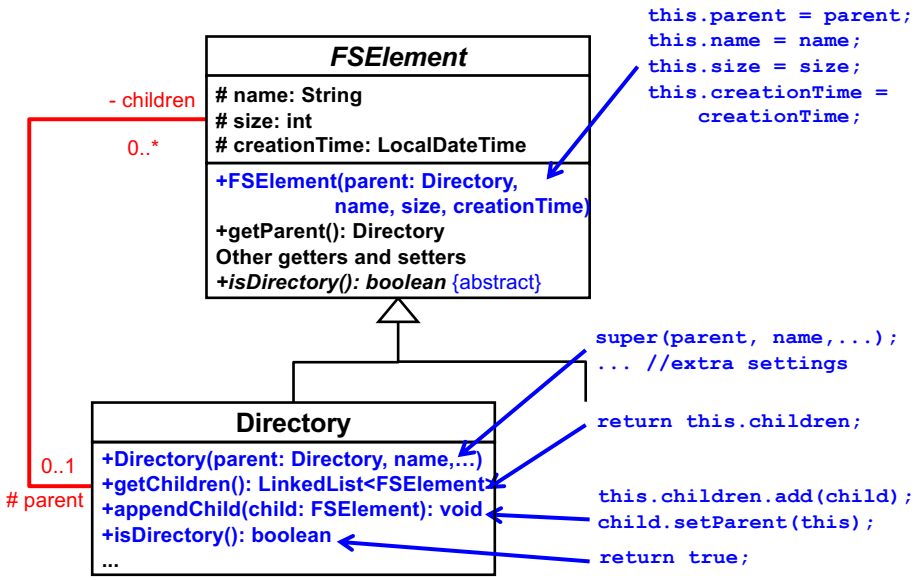
- How to design directory-to-directory structures?
- How to design file-to-directory structures?

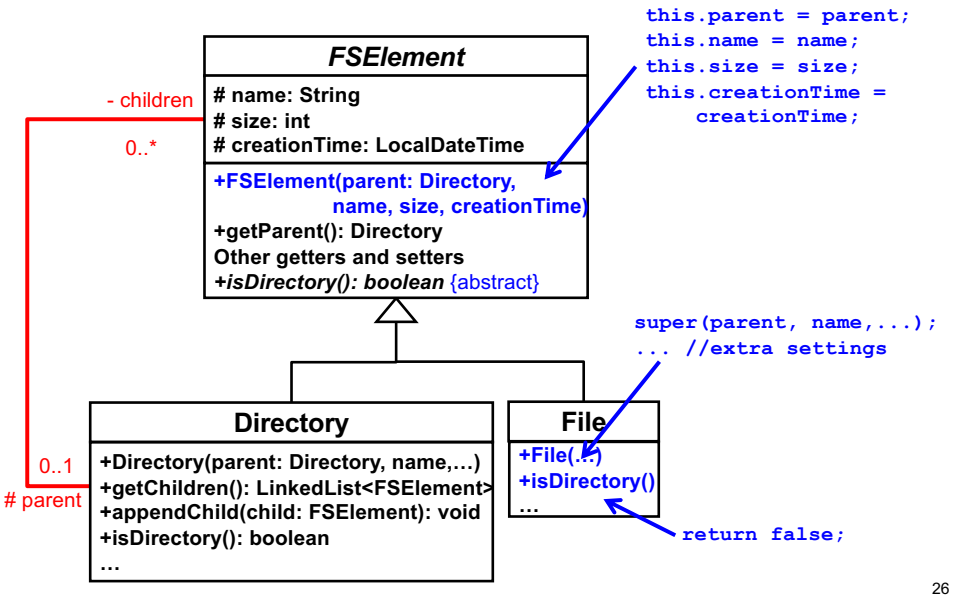


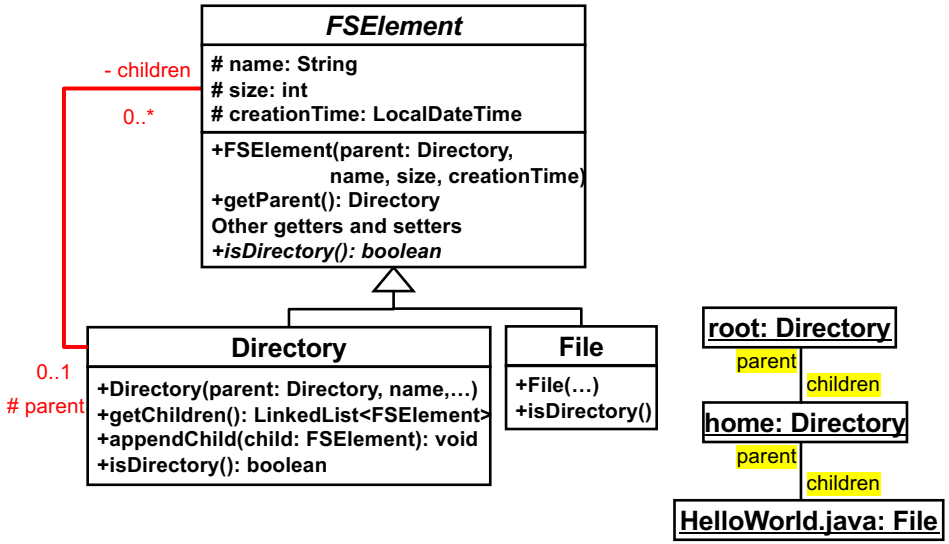
Composite Design Pattern











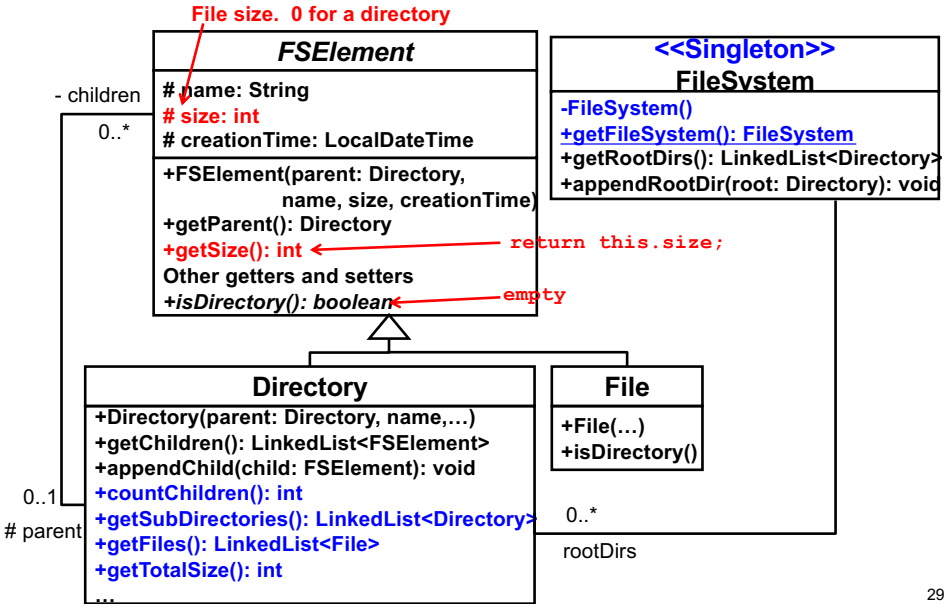

```
Directory root = new Directory(null, "root", ...);
Directory home = new Directory(root, "home", ...);

File file = new File(null, "HelloWorld.java", ...);
home.appendChild(file);

root.getChildren(); home.getChildren();
```

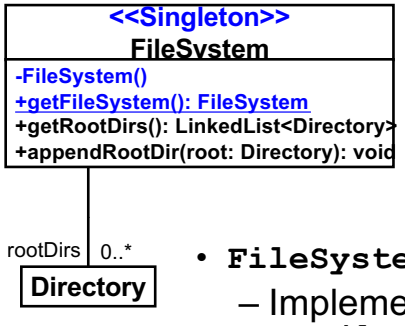

Benefits of *Composite*

- Client code of a tree structure can treat individual objects and compositions of objects uniformly.
  - Objects → files and directories
  - Compositions of objects → directories that compose/contain subdirectories and files
  - "Treating…uniformly" means…
    - Having common data fields and methods (across files and directories) in a super class, and
    - Performing polymorphism on files and directories.

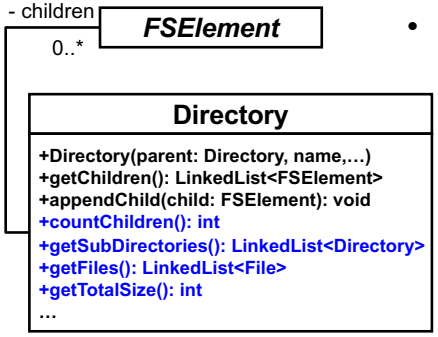
```
- Directory dir = ...;
  for( FSElement fsElement: dir.getChildren() ){
      fsElement.getName();
      fsElement.getSize();
      fsElement.getParent().getName(); }
```

HW 6: Implement This



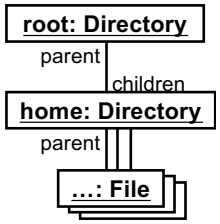


- `FileSystem` (*Singleton* class)
  - Implement a static factory method: `getFileSystem()`
  - Define a private constructor
  - Assume this file system can have multiple "drives" (multiple tree structures) in it.
    - c.f., C, D and E drives in Windows



- `Directory`
  - `LinkedList<FSElement> children`: data field to reference files and subdirectories in a directory
  - `countChildren()`: returns the number of files and subdirectories in a directory.
  - `getTotalSize()`: returns the total disk consumption by all the files and subdirectories under a directory
    - e.g., Call `getTotalSize()` recursively on all subdirectories

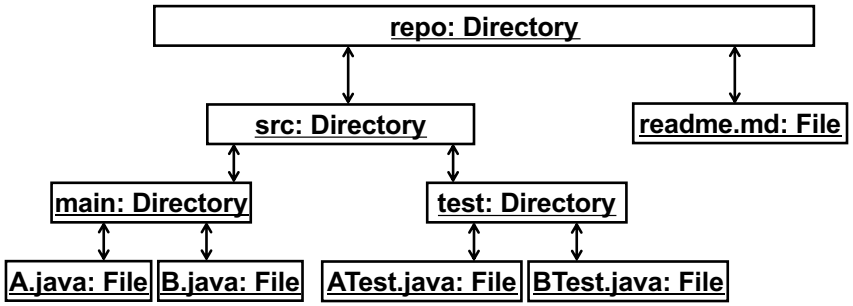
```
Directory root = new Directory(null, "Root", ...);
Directory home = new Directory(root, "home", ...);
File file = new File(home, "...", ...);
File file = new File(home, "...", ...);
File file = new File(home, "...", ...);

root.getTotalSize();
```



- You can add any extra methods in any classes as you like/need.
  - The class diagram in a previous slide is not complete.
    - e.g., `isDirectory()` in `Directory`

Unit Testing

- Use this file system structure in your test cases.
  - Create this file system structure as a test fixture.
  - Assign values to data fields (size, etc) as you like.



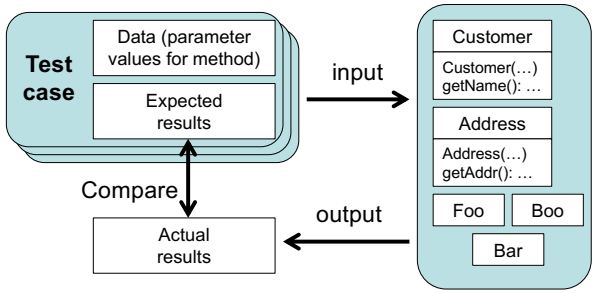
- Test each public method of `Directory`, `File` and `FileSystem` with `DirectoryTest`, `FileTest`, `FileSystemTest`

- In addition…
  - `DirectoryTest`
    - Implement an equality-check for each `Directory` instance by calling getter methods of `Directory`
      - Use `name`, `size` and `creationTime` for equality check.
      - c.f. previous HW, in which you did equality-check for different `Car` instances
  - `FileTest`
    - Implement an equality-check for each `File` instance by calling getter methods of `File`
      - Use `name`, `size` and `creationTime` for equality check.


```
• public DirectoryTest{
    ...
    private String dirToStringArray(Directory d){
      String[] dirInfo = {
        d.getName(), d.getSize(), ...};
      return dirInfo; }

    @Test
    public void verifyDirectoryEqualityRoot(){
      String[] expected = {..., ..., ...};

      Directory actual = ...;
      assertArrayEquals(expected,
                        dirToStringArray(actual) ); }

    @Test
    public void verifyDirectoryEqualityHome (){ ... }

    ...
  }
```


Test Fixtures

- Fixture
  - An instance of a class under test
  - An instance of another class that the class under test depends on
  - Input data
  - Expected result(s)
  - Set up of a file(s) and other resources
    - e.g., Socket
  - Set up of external systems/frameworks
    - e.g. Database, web server, web app framework, emulator (e.g. Android emulator)



Setting up Fixtures

- Class under test

```
public class Calculator{
 public int multiply(int x, int y){
   return x * y;
 }
 public float divide(int x, int y){
   if(y==0) throw
    new IllegalArgumentException(
     "division by zero");
   return (float)x / (float)y;
 }
]
```

- Test class

```
public class CalculatorTest{
 @Test
 public void multiply3By4(){
   Calculator cut = new Calculator();
   float actual = cut.multiply(3,4);
   float expected = 12;
   assertEquals(expected, actual); }

 @Test
 public void divide3By2(){
   Calculator cut = new Calculator();
   float actual = cut.divide(3,2);
   float expected = 1.5f;
   assertEquals(expected, actual)); }

 @Test
 public void divide5By0(){
   Calculator cut = new Calculator();
   try{
     cut.divide(5, 0);
     fail("Division by zero"); }
   catch(IllegalArgumentException ex){
     assertEquals("division by zero",
                  ex.getMessage() ); } } }
```

Setting up fixtures
-- inline setup

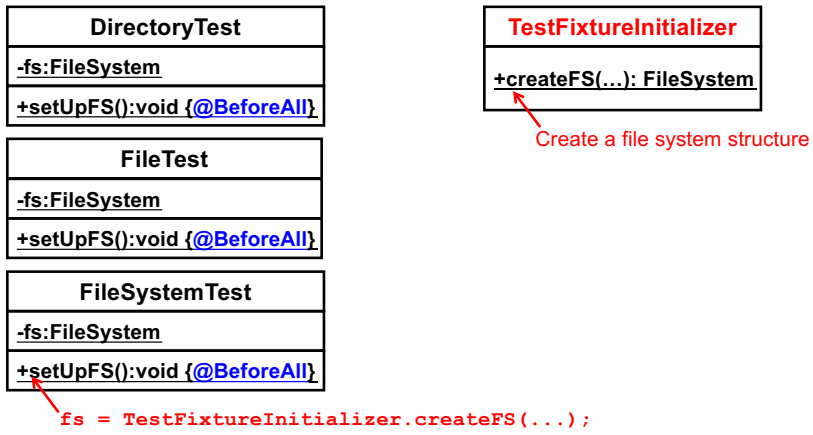
Implicit Setup

```
import org.junit.jupiter.api.BeforeAll;
import org.junit.Jupiter.api.AfterAll;

public class CalculatorTest{
    private static Calculator cut;

    @BeforeAll
    public static void setUp(){
        cut = new Calculator(); }

    @Test
    public void multiply3By4(){
        float actual = cut.multiply(3,4);
        float expected = 12;
        assertEquals(expected, actual); }

    @Test
    public void divide3By2(){
        float actual = cut.divide(3,2);
        ... }

    @Test
    public void divide5By0(){
        ... }

    @AfterAll
    public static void doSomething(){...}  }
```

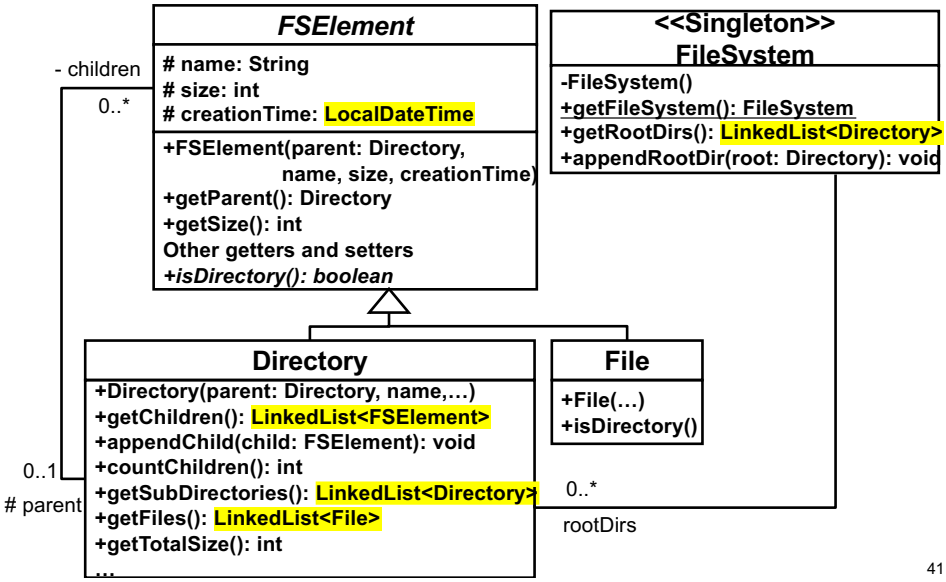

- Implicit setup makes a test class less redundant.
- Flow of execution
  - @BeforeAll setUp()
  - @Test multiply3By4()
  - @Test divide3By2()
  - @Test divide5By0()
  - @AfterAll doSomething()
  - The @BeforeAll method runs before all test methods.
  - The @AfterAll method runs after all test methods.
  - JUnit may run the test methods in a different order from their ordering in source code.

Preparing a Fixture for Multiple Test Classes

- Define a separate class that is responsible for creating a fixture(s).
  - Make sure to take this approach in your HW solution.

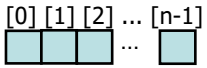


Notes: `LocalDateTime` and `LinkedList`

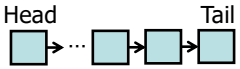


Just in Case: Major Collection Types in Java

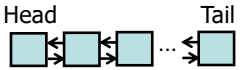
- List
  - Orders elements with integer index numbers.
  - Offers index-based random access.
  - Can contain duplicate elements.



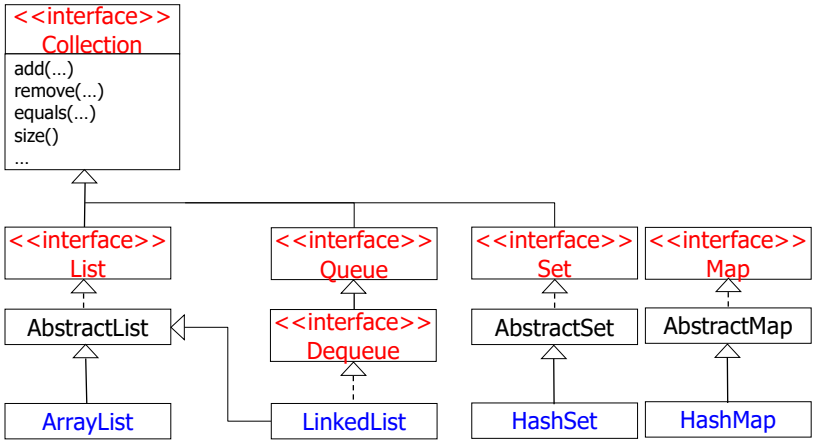
- Queue
  - Orders elements with links.
  - Offers FIFO (First-In-First-Out) access.
  - Can contain duplicate elements.



- Dequeue
  - Stands for "Double Ended QUEUE" (pronounced "deck").
  - Orders elements with links.
  - Offers both FIFO and LIFO (Last-In-First-Out) access.
  - Can contain duplicate elements.

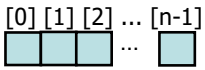


- Set
  - Contains non-duplicate elements without an order.
- Map
  - Contains key-value pairs (w/ non-duplicate keys) without an order.

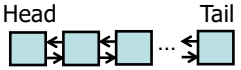


ArrayList v.s. LinkedList

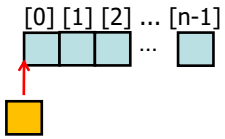
- ArrayList
  - Array-based implementation of the List interface



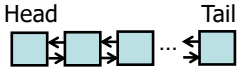
- LinkedList
  - Link-based implementation of the List and Deque interfaces



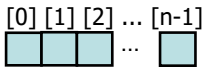
- ArrayList
  - Array-based impl of the List interface
  - Fast index-based access
  - Slow insertion and removal of non-tail elements
    - Fast insertion and removal of the tail element



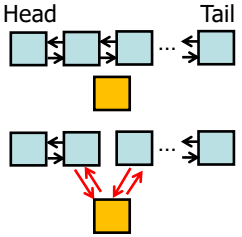
- LinkedList
  - Link-based impl of the List and Deque interfaces



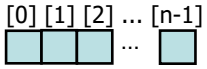
- ArrayList
  - Array-based impl of the List interface
  - Fast index-based access
  - Slow insertion and removal of non-tail elements
    - Fast insertion and removal of the tail element



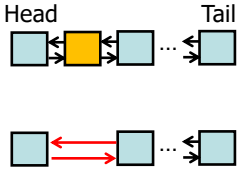
- LinkedList
  - Link-based impl of the List and Deque interfaces
  - Fast insertion and removal of elements



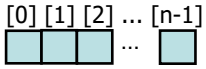
- ArrayList
  - Array-based impl of the List interface
  - Fast index-based access
  - Slow insertion and removal of non-tail elements
    - Fast insertion and removal of the tail element



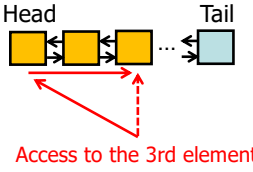
- LinkedList
  - Link-based impl of the List and Deque interfaces
  - Fast insertion and removal of elements



- ArrayList
  - Array-based impl of the List interface
  - Fast index-based access
  - Slow insertion and removal of non-tail elements
    - Fast insertion and removal of the tail element



- LinkedList
  - Link-based impl of the List and Deque interfaces
  - Fast insertion and removal of elements
  - Slow index-based access for “middle” elements.



- Use ArrayList
  - If you often need to access “middle” elements with their index numbers
    - i.e., if index numbers mean something important/special.
- Use LinkedList
  - If you often need to insert/remove elements.
  - If you don’t need index-based element access.
- Both yield the same performance for an element traversal (i.e. sequential element access).
  - `for(… element: anArrayList){…}`
  - `for(… element: aLinkedList){…}`

- ArrayList
  - Not that great, performance-wise, in multi-threaded programs.
    - Hard to make element traversal concurrent/parallel.
- LinkedList
  - More friendly for concurrency/parallelism.