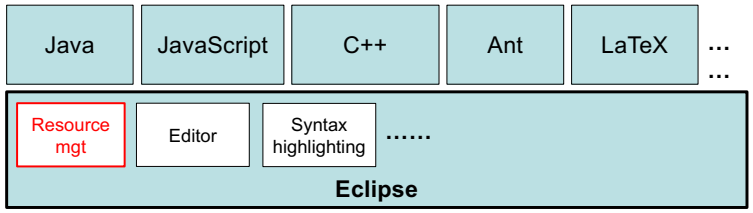
Another Example of *Factory Method* in Integrated Dev Environments (IDEs)

- An IDE (e.g., Eclipse and IntelliJ) consists of various components (or plugins)
  - Java editor, JavaScipt editor, C++ editor, Ant build script editor, CSS editor, Java syntax highlighter, JUnit test result viewer, etc.

- Each IDE component uses resources.
  - Resources:
    - Programs (e.g., Java, JavaScript, C++, etc.),
    - XML files (e.g., build.xml for Ant, web.xml for Servlet WAR),
    - HTML files, CSS files, etc.
  - Java editor and syntax highlighter use Java programs.
  - C++ editor and syntax highlighter use C++ programs.
  - Ant editor and syntax highlighter use build scripts.

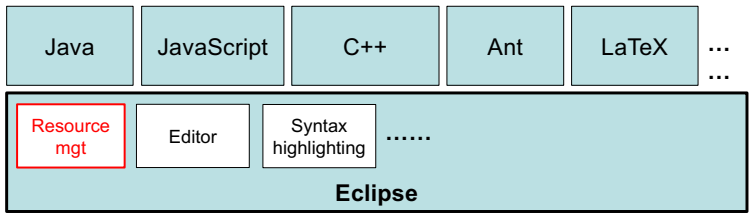
- A modern IDE implements resource mgt. in its foundation API.
  - because it is a common, required feature in many IDE components.

- Resource management
  - Creating, opening and closing *resources* used by IDE components
  - Saving resources.
  - Renaming resources.
  - Moving resources (changing their locations).
  - Exporting resources to remote source code repositories (e.g. GitHub).

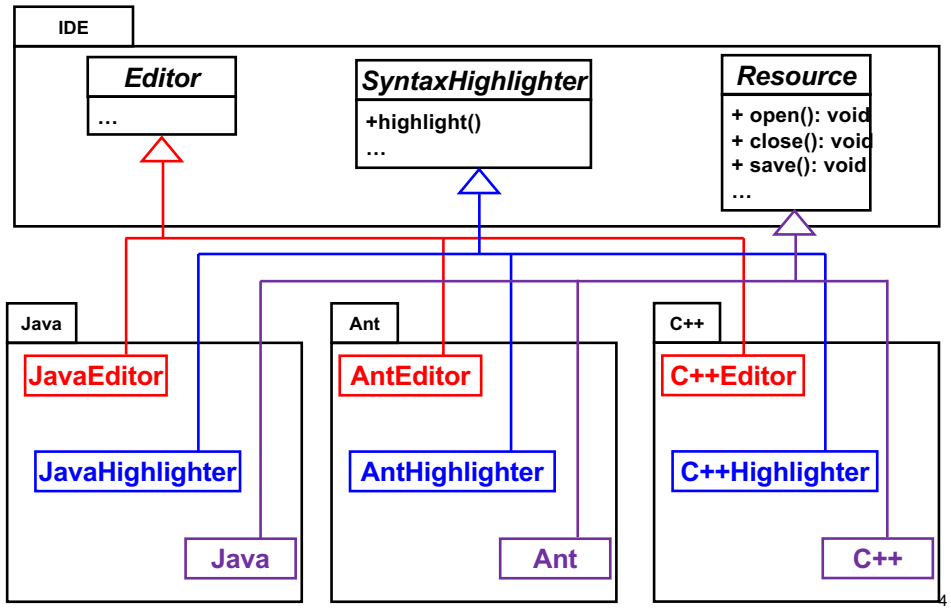


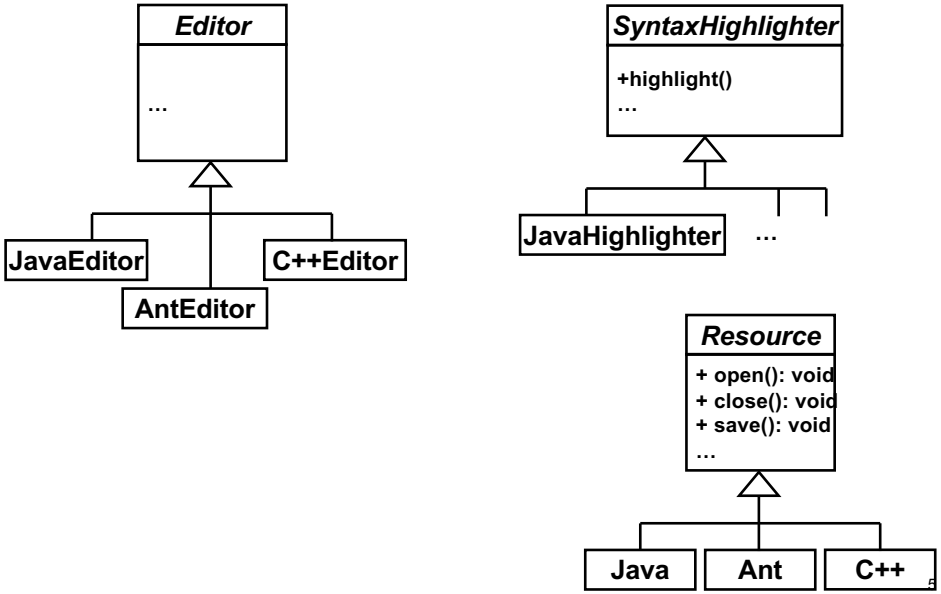
- The foundation API is intended to
  - Help developers implement IDE components as quickly as possible
    - By providing resource mgt as a reusable functionality
  - Provide a consistent resource mgt. behavior among different IDE components.

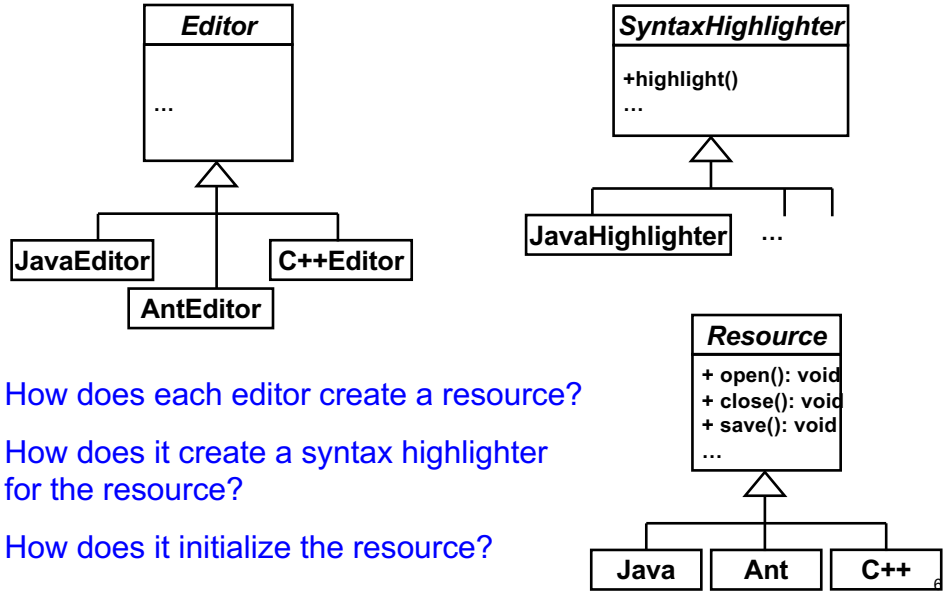
- Now, let's think of implementing the ***creation*** of resources.



Editors, Syntax Highlighters and Resources





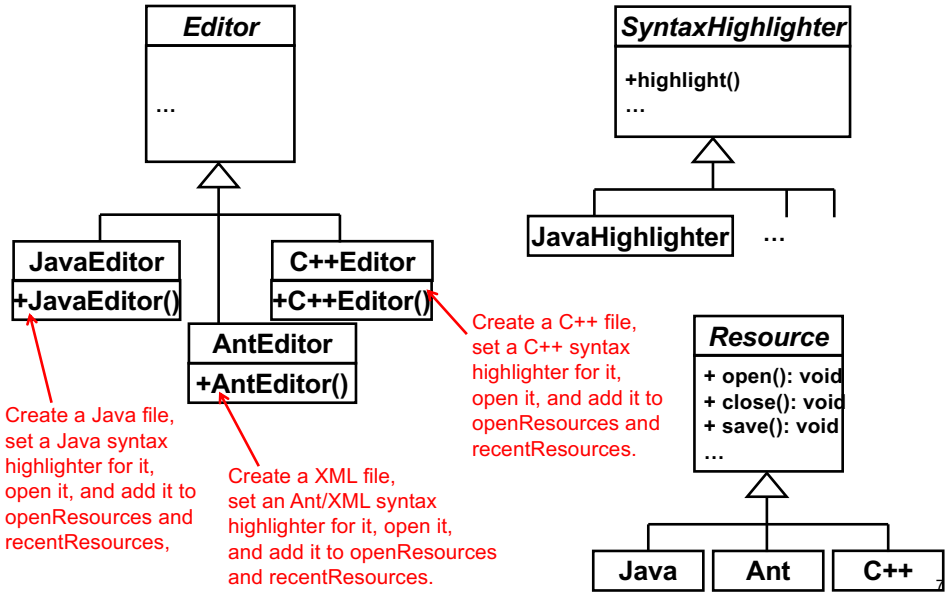


How does each editor create a resource?

How does it create a syntax highlighter for the resource?

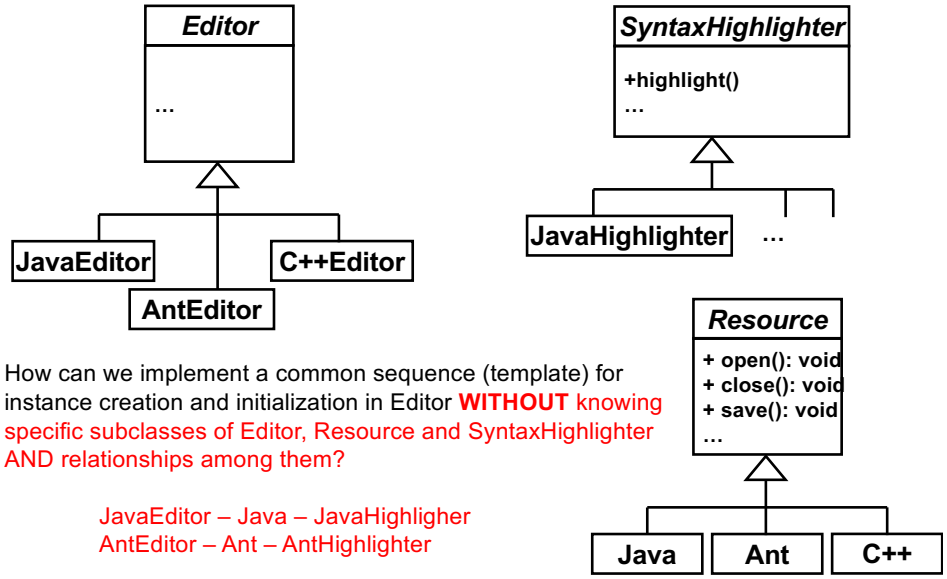
How does it initialize the resource?

If You don't Use *Factory Method…*



- This IDE
  - Is not that friendly for IDE component developers.
    - Does not provide a common sequence (or skeleton/template) to create and initialize resources.
    - Requires developers to write redundant code for their editors.
  - Can be more developer friendly
    - By offering a common sequence (or template) to create and initialize a resource in `Editor`.
      - Does not have to require developers to write redundant code.

Dilemma

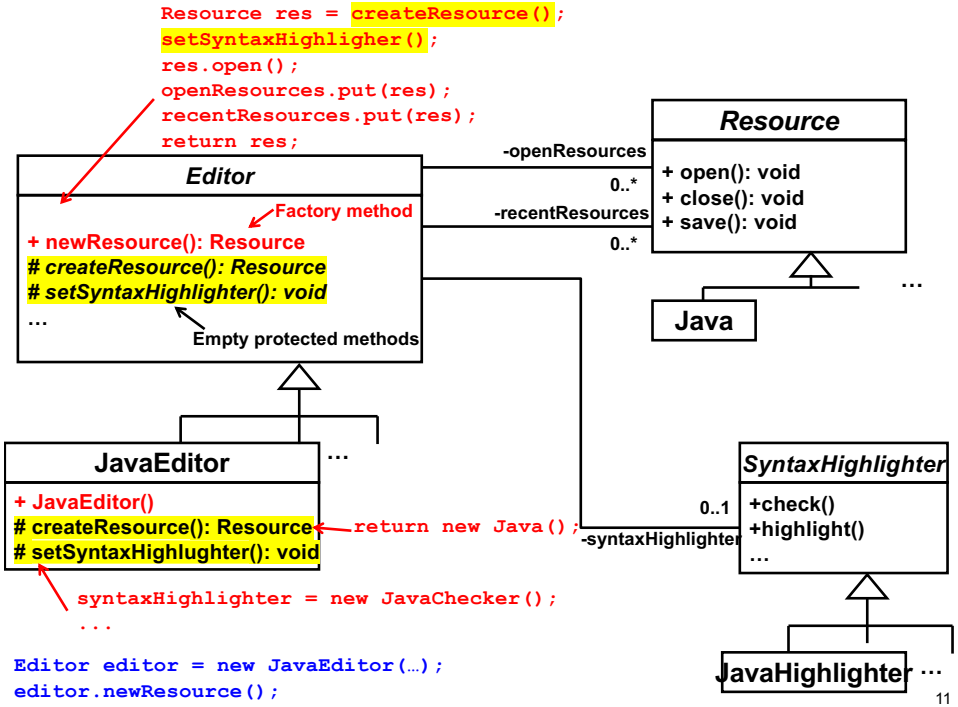


How can we implement a common sequence (template) for instance creation and initialization in Editor **WITHOUT** knowing specific subclasses of Editor, Resource and SyntaxHighlighter AND relationships among them?

JavaEditor – Java – JavaHighligher
AntEditor – Ant – AntHighlighter

Approach the Dilemma w/ *Factory Method*

- Define a *factory method* in `Editor`.
- Have it implement a common sequence (or template) for instance creation and initialization *with an empty protected method(s)*.



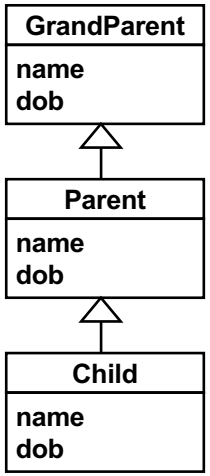
Benefits of *Factory Method*

- This IDE
  - Can define a common sequence (or a template) for instance creation and initialization
  - Allows individual editors to reuse it.
    - Less code redundancy in implementing editors
    - Can "force" every single editor to follow a consistent behavior when it creates a new resource.
  - Does not have to know editor-resource-syntax highlighter relationships.

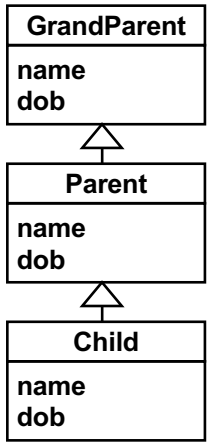
– Can be independent (or de-coupled) from individual IDE components.
  - Allows IDE components to be pluggable to the framework.

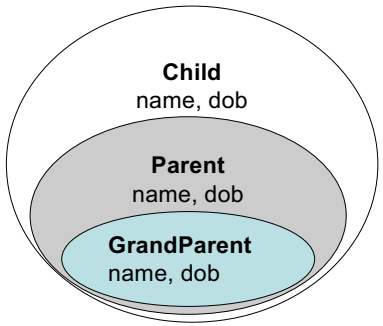
Implementing a Hierarchy of Objects with a Recursive Association

Wrong Design



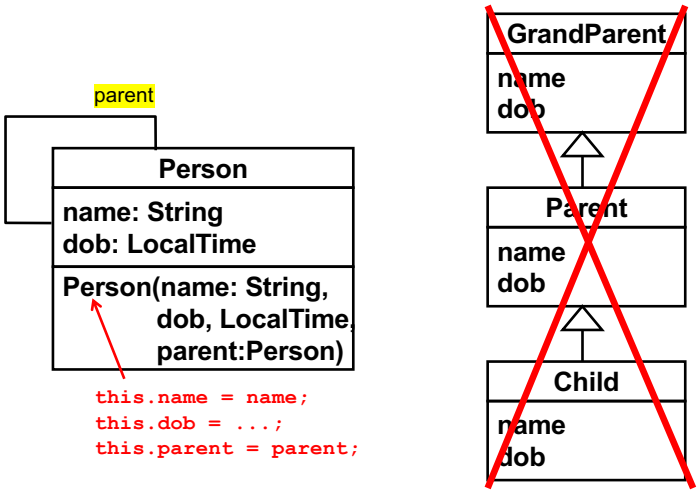
- A classical design error in/since mid ’80.
  – A parent “inherits” a grand parent’s data fields and methods.
  – A child “inherits” a parent’s data fields and methods.
- Found in an OOP textbook published in 2009 (!)





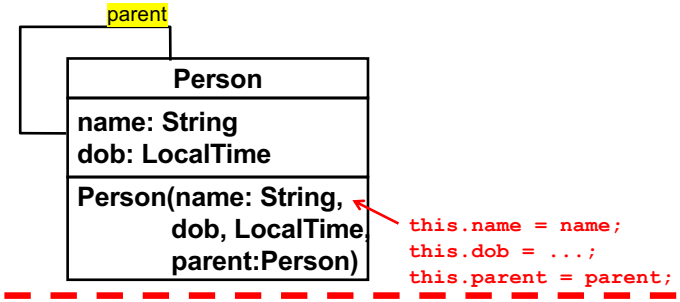
- A parent has two “name” data fields and two “dob” data fields!
- A child has three “name” data fields and three “dob” data fields!!

Right Design

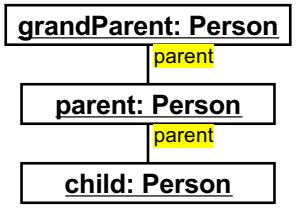


- Use a recursive association rather than class inheritance

Class diagram



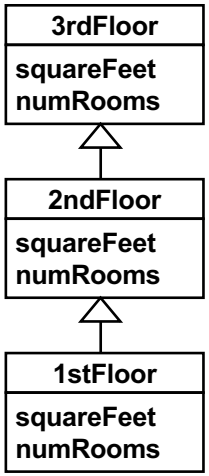
Object diagram

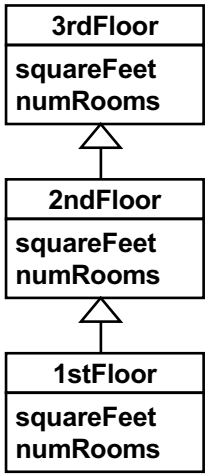


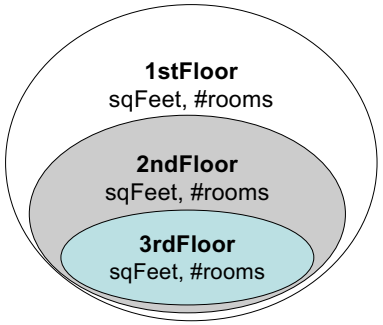

```
• Person grandParent = new Person("grandpa", ..., null);
  Person parent      = new Person("dad", ..., grandParent);
  Person child       = new Person("me", ..., parent);

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

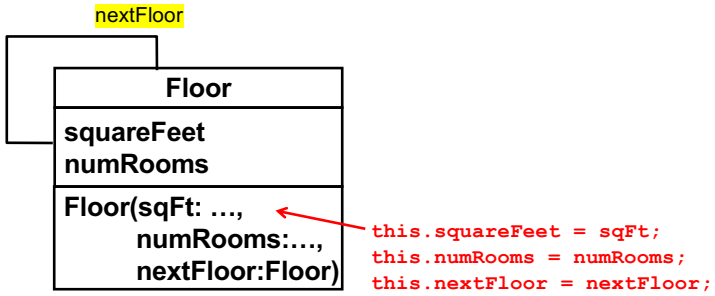

Another Example



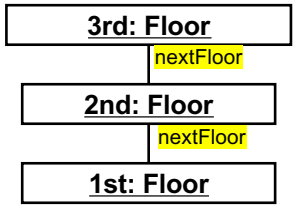




Class diagram



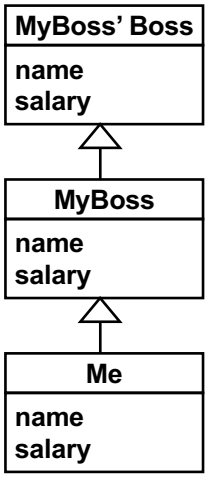
Object diagram



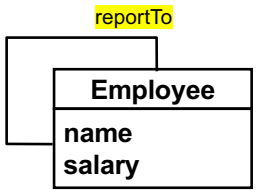

- ```
  Floor 3rd = new Floor(500, 2, null);
  Floor 2nd = new Floor(700, 3, 3rd);
  Floor 1st = new Floor(1000, 5, 2nd);

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

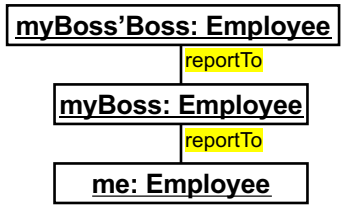

One More Example



Class diagram



Object diagram




- ```
  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();
  ```

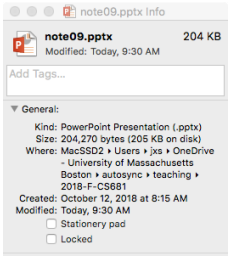

Composite Design Pattern

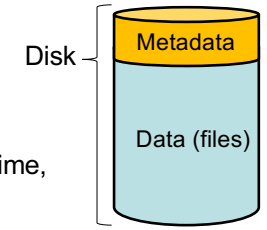
Composite Design Pattern

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

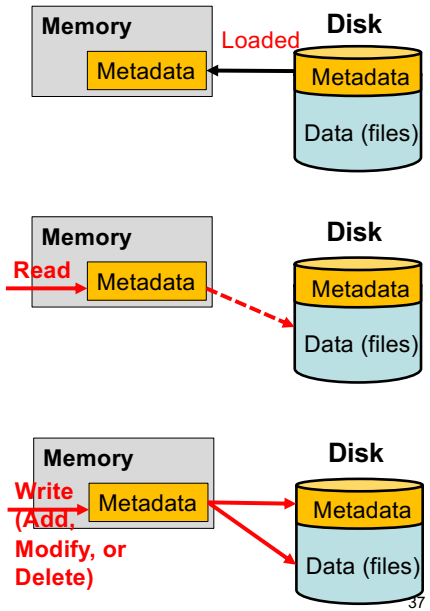
Example: Metadata Mgt in File Systems

- 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 (i.e., 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



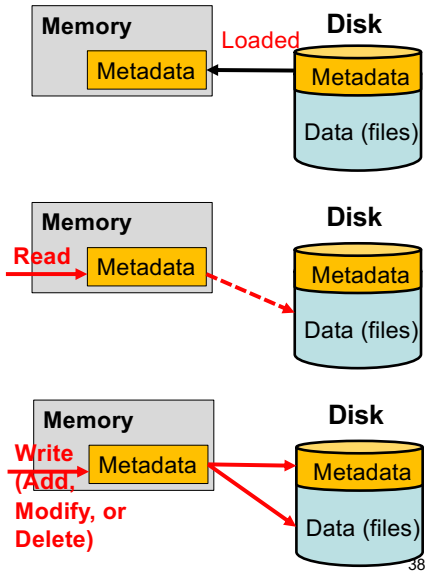


- An OS loads metadata to the main memory during its bootup process.
- Applications access (read and write) files through their metadata.
  - Read a file's metadata
  - Read a file's content
  - Add/store a new file
  - Modify a file's metadata and/or content.
  - Delete a file.



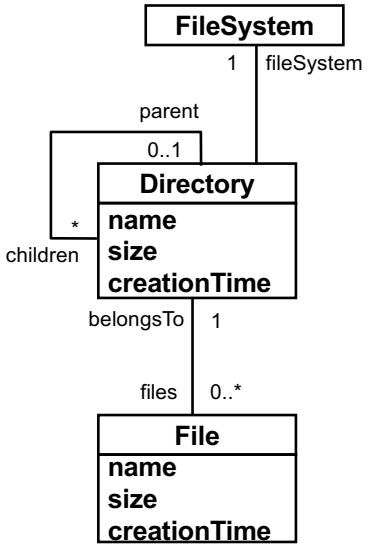
Metadata in a File System

- 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.



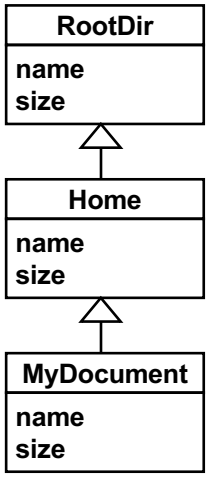
Requirements for Metadata Design

- 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)

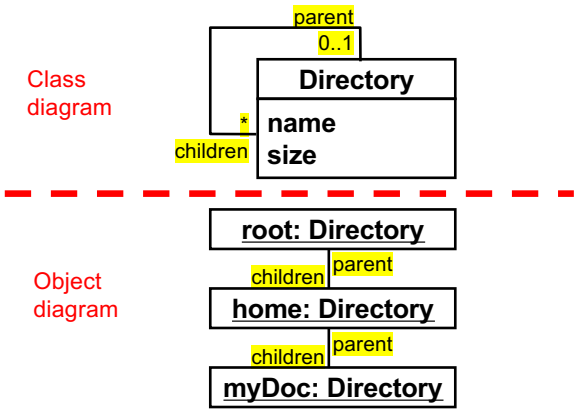


- A file system consists of directories and files.
- Each file exists in a particular directory.
- Each directory can contain multiple files.
- Directories form a tree structure.
  - Every directory has its parent directory, except the root directory.
  - Each directory can have multiple sub directories.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

Don't Do This.

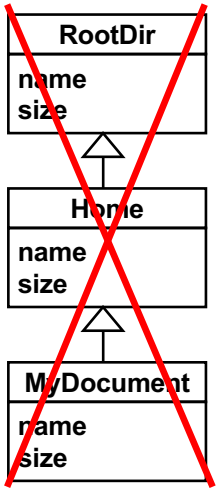


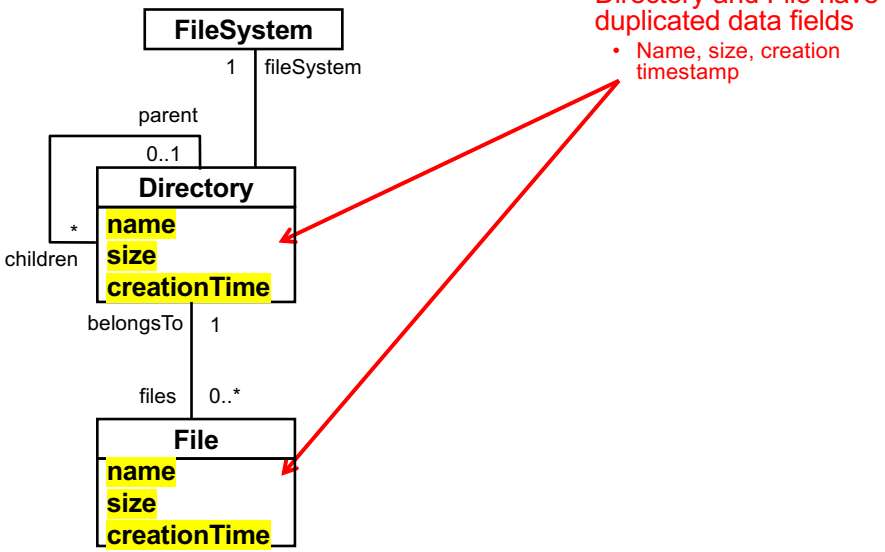
Do This.



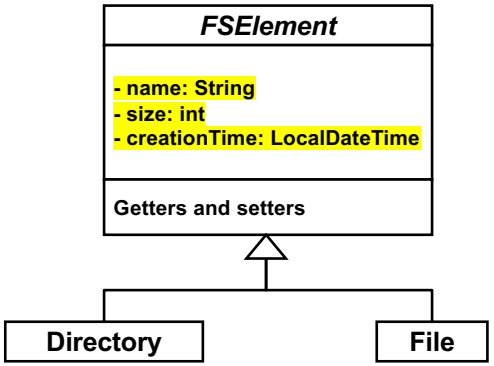

```
Directory root =  new Directory(null, ...);
Directory home =  new Directory(root, ...);
Directory myDoc = new Directory(home, ...);

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

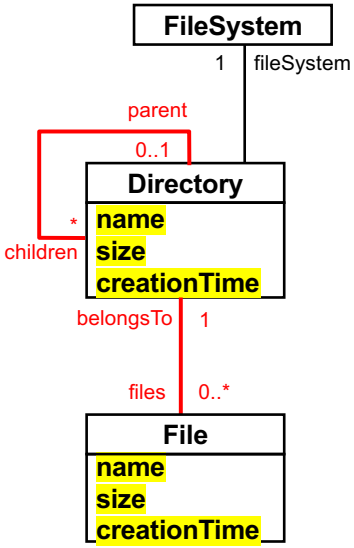




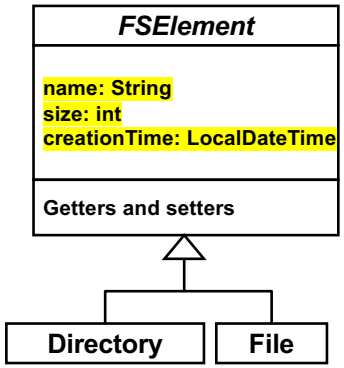
– Directory and File have duplicated data fields
  - Name, size, creation timestamp



- 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