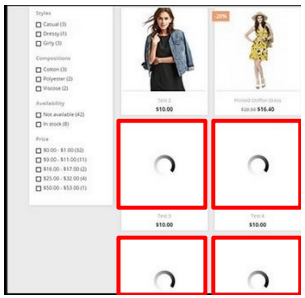
Proxy Design Pattern

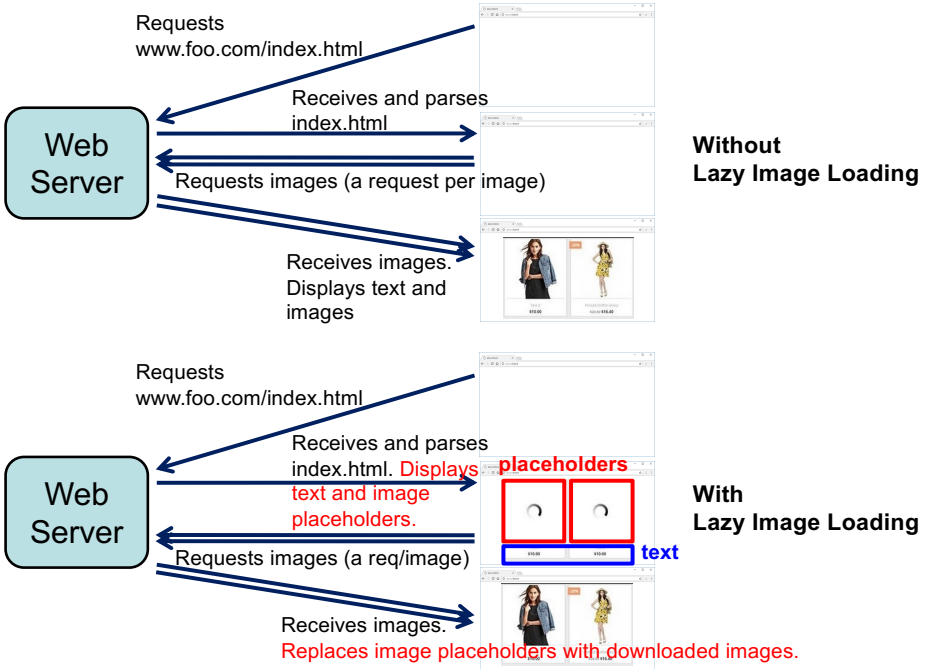
Proxy Design Pattern

- Intent
  - Provide a surrogate (or placeholder, or mock) for another object to control access to it.

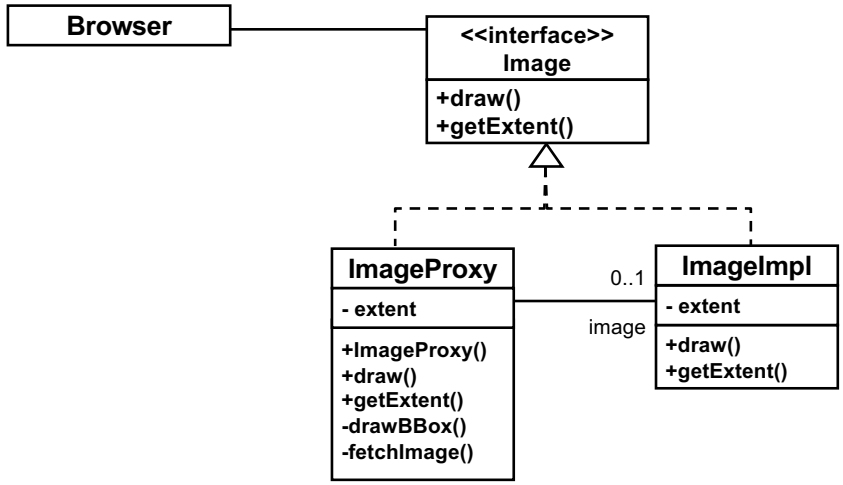
An Example: Lazy Image Loading in a Web Browser

- When an HTML file contains an image(s), a browser
  - Displays a bounding box (placeholder) for each image first
    - Until the browser fully downloads the image.
      - Most users are not patient enough to keep watching a blank browser window until all text and images are downloaded and displayed.
  - Replaces the bounding box with the real image.





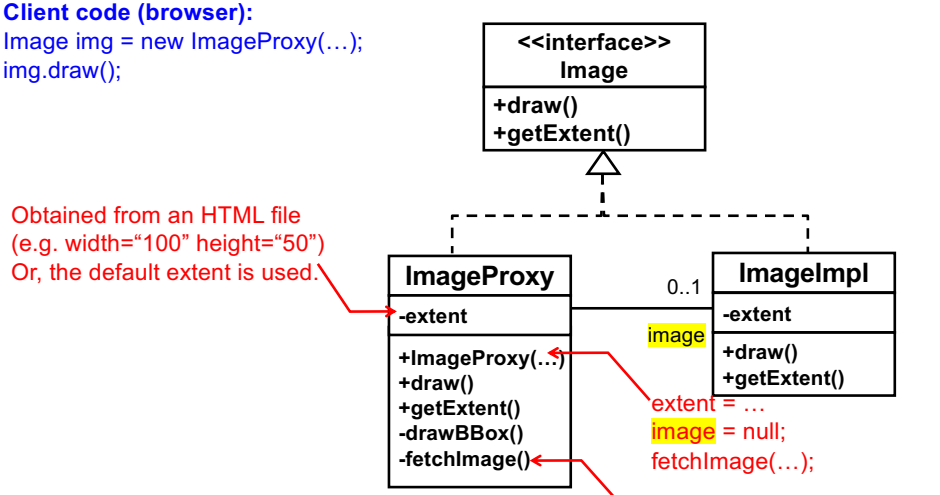
Client



Proxy Class

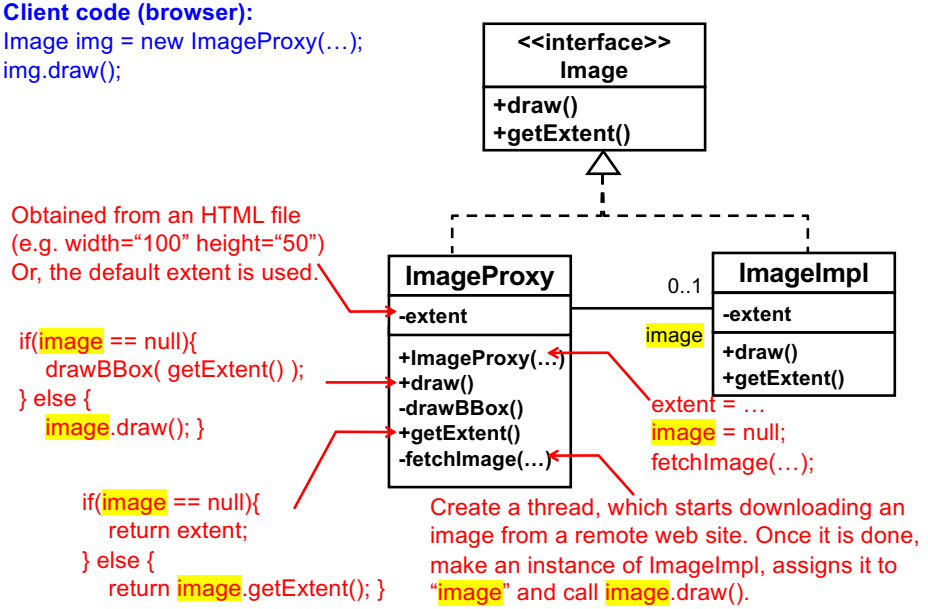
Client code (browser):

```
Image img = new ImageProxy(…);
img.draw();
```



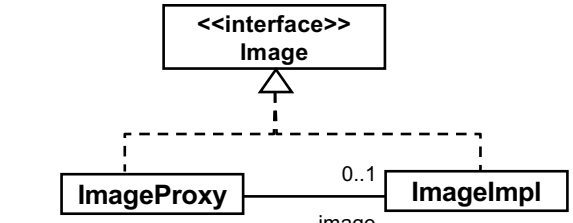
Client code (browser):

```
Image img = new ImageProxy(…);
img.draw();
```



What's the Point?

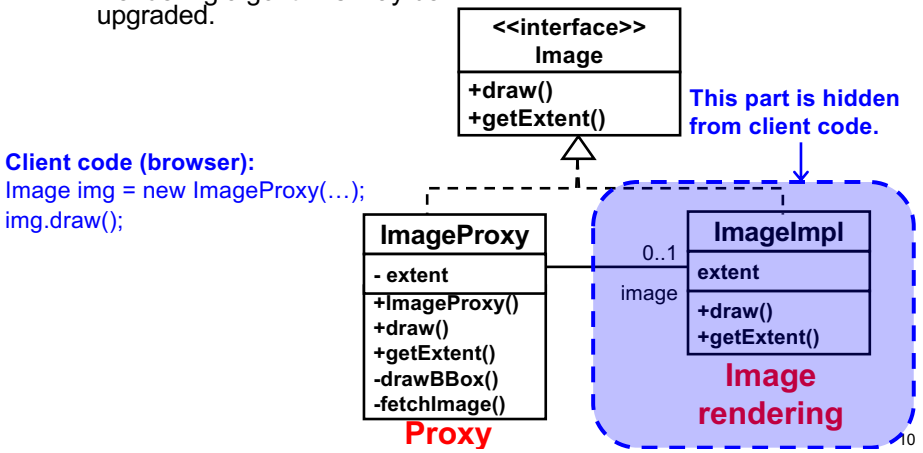
- Separate *bounding box placement (lazy image loading)* and *image rendering*.
  - Make the two concerns independent with each other
    - Separation of concerns to improve maintainability



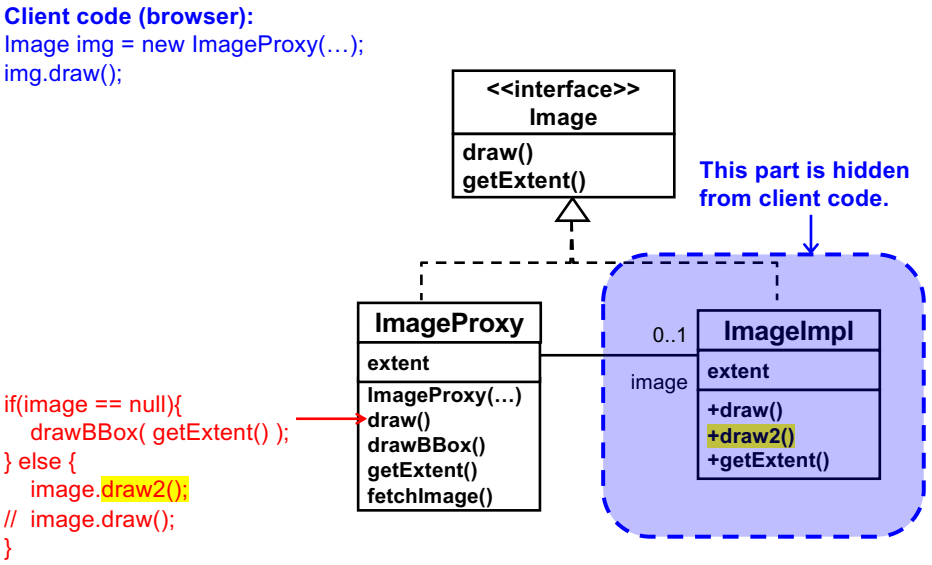
Lazy image loading **Image rendering**

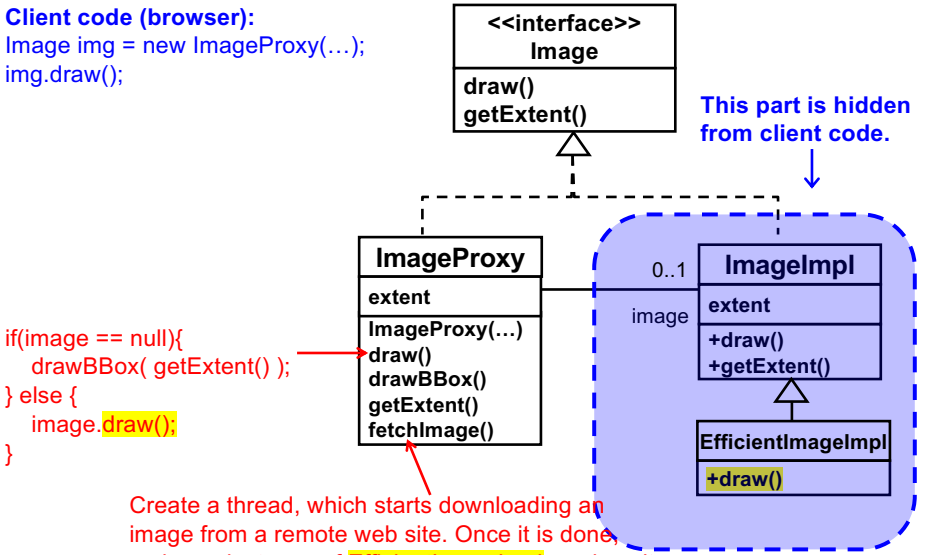
- Separation of concerns improves maintainability
- When a change is made on bounding box placement, you can leave image rendering as it is.
  - The look-and-feel of a bounding box may change.
  - Concurrency policy may change.
- When a change is made on image rendering, you can leave bounding box placement as it is.
  - New image formats may be introduced.
  - Image rendering algorithms may be upgraded.

- *Proxy* can hide image rendering from its client (browser).
  - The client uses (or faces) ImageProxy, not ImageImpl.
  - When a change is made on image rendering, you don't have to change client code.
    - New image formats may be introduced.
    - Rendering algorithms may be upgraded.



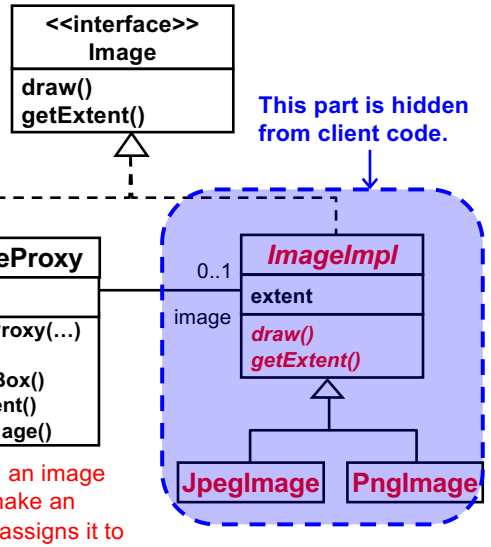
Supporting a New Rendering Algorithm





Supporting Multiple Image Formats

Client code (browser):
Image img = new ImageProxy(…);
img.draw();

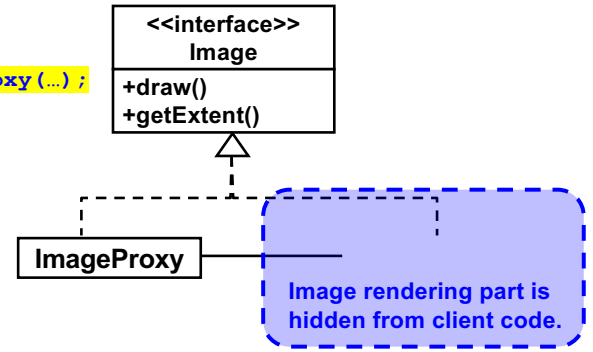


Create a thread, which starts downloading an image from a remote web site. Once it is done, make an instance of JpegImage or PngImage, and assigns it to "image" and call image.draw().

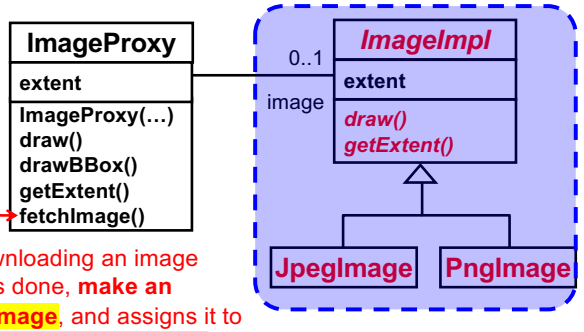
Two Possible Design Improvements

Client code (browser):

```
Image img = new ImageProxy(…);
img.draw();
```



- Client code
  - Doesn't have to know the details about image rendering
  - Does need to know about `ImageProxy` (i.e., need to know that *Proxy* is used to draw images).
  - Actually doesn't have to know whether or not *Proxy* is used (i.e., whether or not lazy image loading is enabled).
    - Let's separate (decouple) `ImageProxy` and its client.



Create a thread, which starts downloading an image from a remote web site. Once it is done, **make an instance of JpegImage or PngImage**, and assigns it to "image" and image.draw(). A conditional comes here.

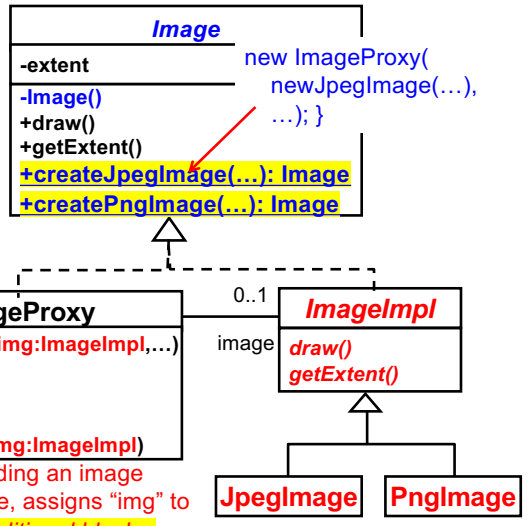
- `ImageProxy`
  - Now needs to know what image formats the browser supports.
  - Actually doesn't have to (want to) know that.
    - Let's separate (decouple) `ImageProxy` from the choice of image formats

One Step Further with *Static Factory Method*

Client code (browser):

Image img =
  Image.createJpgImage("…jpg", …);
img.draw();

Client no longer need to face ImageProxy. It faces Image now.

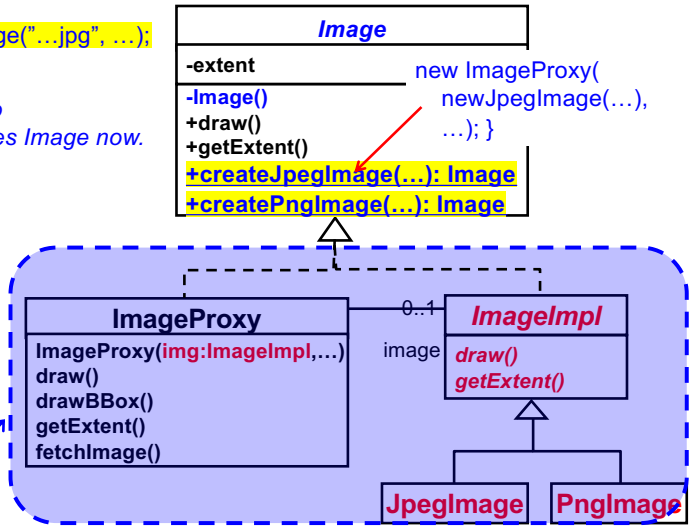


Create a thread, which starts downloading an image from a remote web site. Once it is done, assigns "img" to "image" and call image.draw(). *No conditional blocks come here. ImageProxy is independent from image format choices and rendering algorithms.*

Client code (browser):

```
Image img =
   Image.createJpgImage("…jpg", …);
img.draw();
```

Client no longer need to face ImageProxy. It faces Image now.

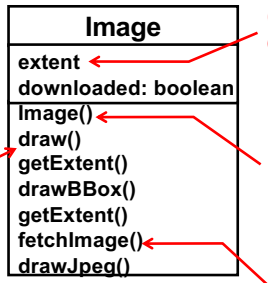


This part is now hidden from client code.
Client code doesn't have to know the details about image loading and image rendering.

What if Everything is Integrated in a Single Class?

Client code (browser):

```
Image img = new Image(…);
img.draw();
```



Obtained from a HTML file (e.g. width="100" height="50") Or, the default extent is used.

```
if(!downloaded){
   drawBBox( getExtent() );
} else {
   if(jpeg is requested){
     drawJpeg();
   }
   …
}
```

```
downloaded=false;
extent = …
fetchImage(…);
```

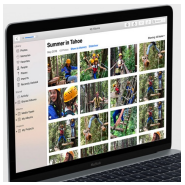
Create a thread, which starts downloading an image from a remote web site. Once it is done, call draw().

Image loading, image formats and image rendering are all mixed up and tangled in a single class, which will not be maintainable.

Better design strategy: Separation of concerns (loosely-coupled design)

Face Detection in Pictures

- Suppose you are implementing an app to organize, edit and analyze pictures.
  - e.g., Photos from Apple

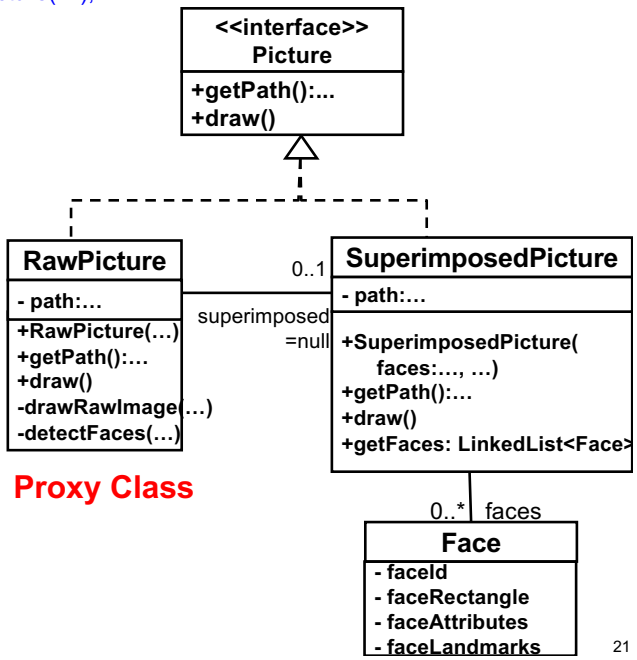


  - The app loads each raw picture and then superimposes a rectangle on a human face by (dynamically) calling an external face detection/recognition API.
    - e.g., APIs from Microsoft, Google, Facebook, etc.

- Some delay is expected to receive a face detection result from an external API.
  - The user is not patient enough to keep watching a blank app window until receiving a detection result.

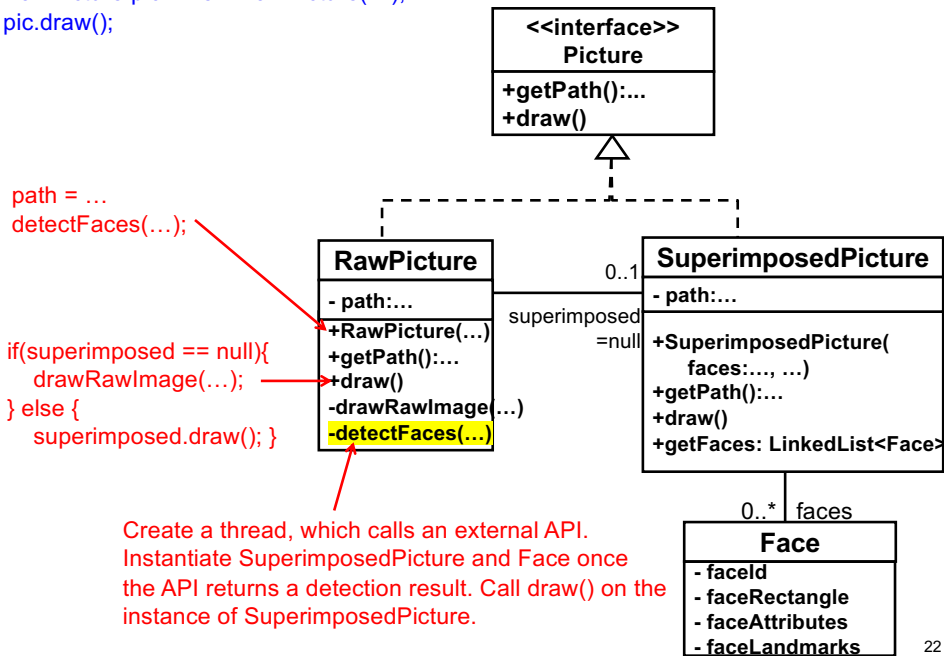
- Lazy loading of detection results
  - Show the user a raw picture first.
  - Call a face detection API in the background
  - Receive a detection result.
  - Replace the raw picture with a superimposed one, which contains a detection result.

Client code (app):
RawPicture pic = new RawPicture(…);
pic.draw();



Proxy Class

Client code (app):
RawPicture pic = new RawPicture(…);
pic.draw();



Create a thread, which calls an external API. Instantiate SuperimposedPicture and Face once the API returns a detection result. Call draw() on the instance of SuperimposedPicture.

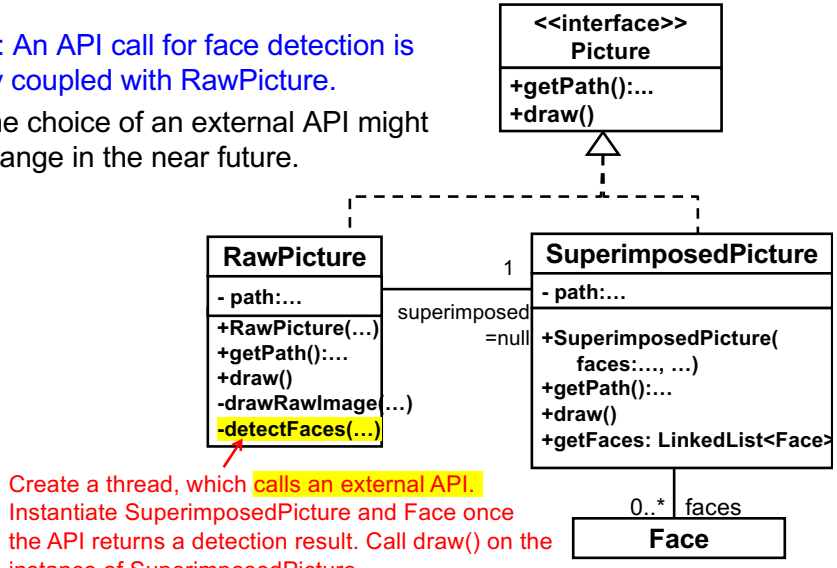
Separation of Concerns

- Lazy loading of face detection results
  - How to display raw pictures
  - How to call an external API and receive a detection result
- Rendering of superimposed pictures
  - How to show face contours
  - What other detection results to display
    - e.g., age, gender, pupil locations, smiling or not, emotion (e.g. happy, angry, sad or surprised)

Further Potential Improvements

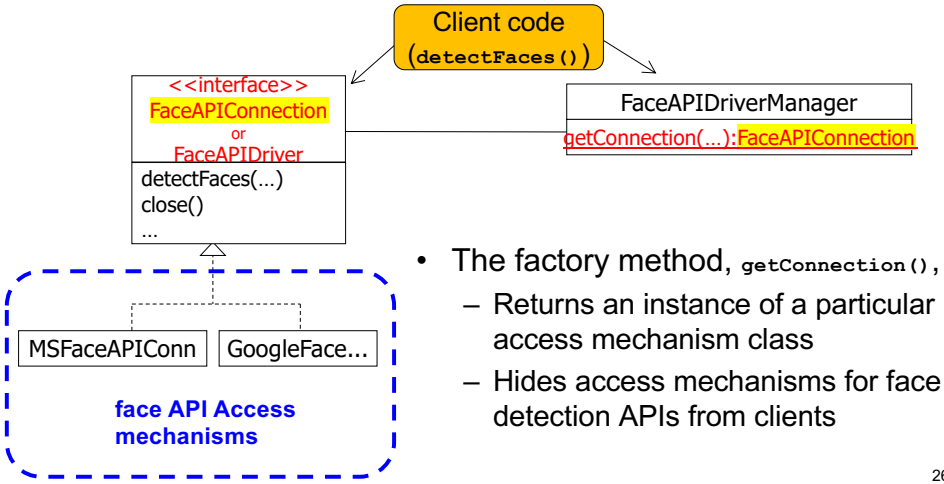
- Lazy loading of detection results is tightly coupled with client
  - You can consider further design extensions that we saw in the previous example.
    - Introducing static factory method(s) in Picture.
- An API call for face detection is embedded (or hard-coded) in RawPicture.
  - The choice of an external API might change in the near future.

- Issue: An API call for face detection is tightly coupled with RawPicture.
  - The choice of an external API might change in the near future.



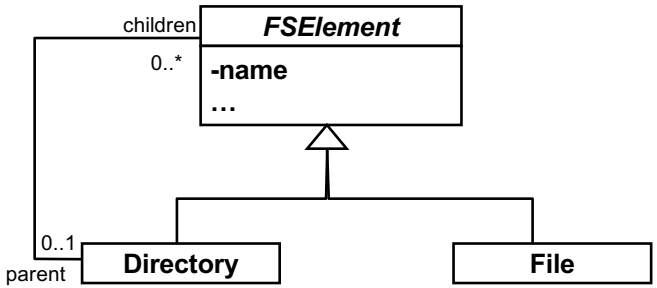
- Have `detectFaces()` obtain an access mechanism to a particular face detection API based on *Iterator*-inspired design.

```
FaceAPIConnection conn =
  FaceAPIDriverManager.getConnection(...);
conn.detectFaces(...);
```



- The factory method, `getConnection()`,
  - Returns an instance of a particular access mechanism class
  - Hides access mechanisms for face detection APIs from clients

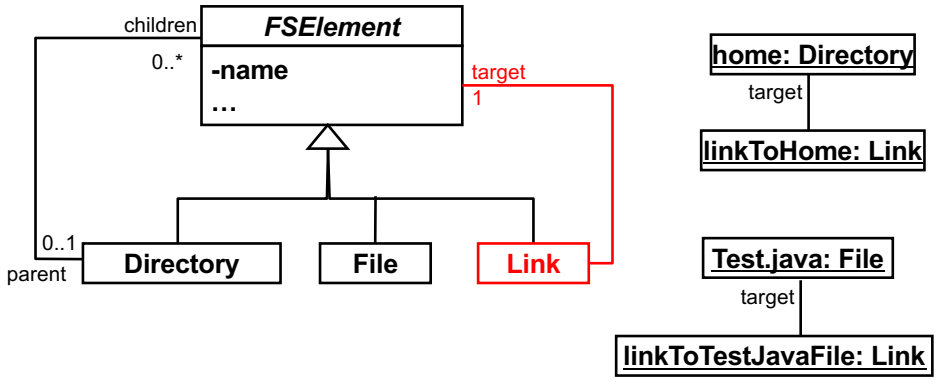
Another Example: Proxies of Files and Directories in File Systems



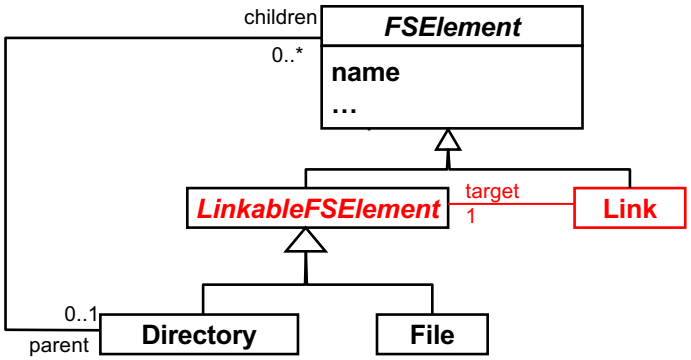
- Let's add symbolic links in addition to files and directories
  - a.k.a. alias (Mac), shortcut (Windows)

```
> ln -s <destination path> <link name/path>
```

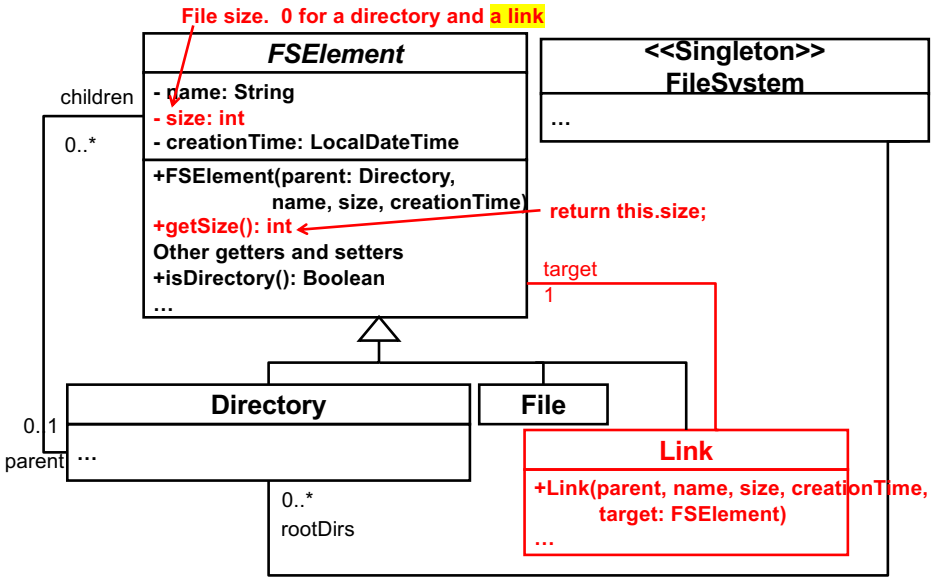
- A link acts as a proxy of a directory or file.
- Let's use the *Proxy* design pattern.



- A link acts as a proxy of a directory or file.
  - A link can act as a proxy of another link too.



HW 7: Implement This



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.