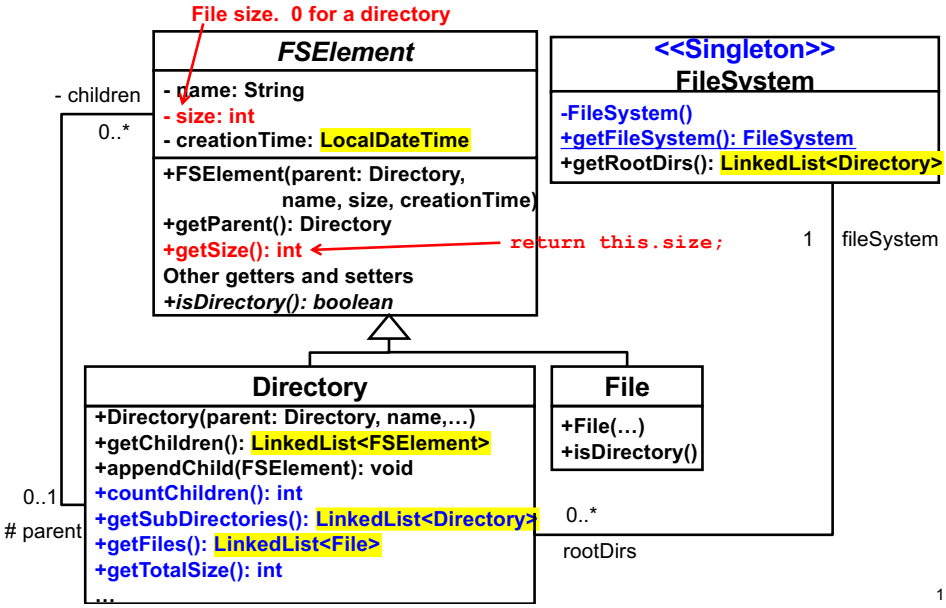
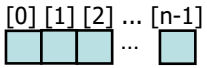
HW 6: Implement This

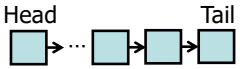


Just in Case: Major Collection Types in Java

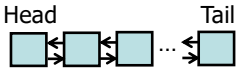
- List
  - Orders elements with integer index numbers.
  - Offers index-based random access.
  - Can contain duplicate elements.



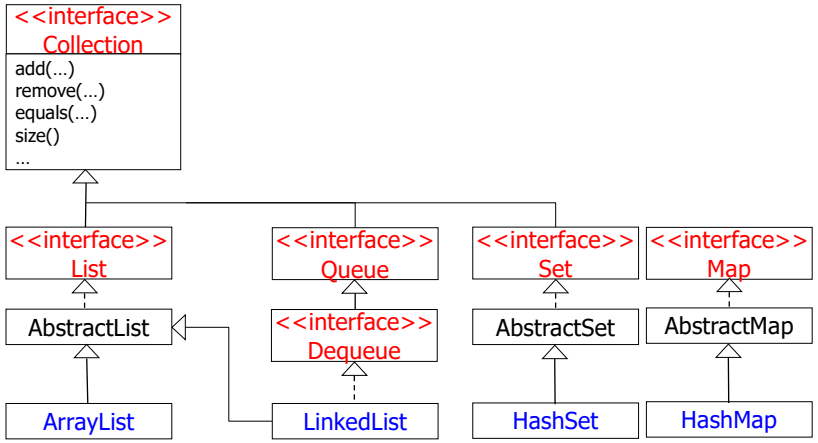
- Queue
  - Orders elements with links.
  - Offers FIFO (First-In-First-Out) access.
  - Can contain duplicate elements.



- 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.

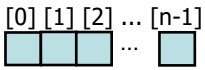


- Set
  - Contains non-duplicate elements without an order.
- Map
  - Contains key-value pairs (w/ non-duplicate keys) without an order.

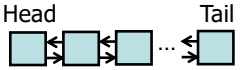


ArrayList v.s. LinkedList

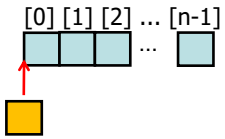
- ArrayList
  - Array-based implementation of the List interface



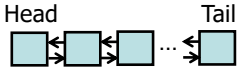
- LinkedList
  - Link-based implementation of the List and Deque interfaces



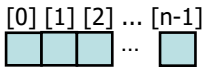
- ArrayList
  - Array-based impl of the List interface
  - Fast index-based access
  - Slow insertion and removal of non-tail elements
    - Fast insertion and removal of the tail element



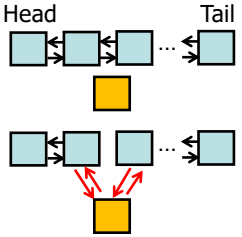
- LinkedList
  - Link-based impl of the List and Deque interfaces



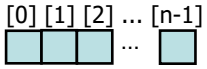
- ArrayList
  - Array-based impl of the List interface
  - Fast index-based access
  - Slow insertion and removal of non-tail elements
    - Fast insertion and removal of the tail element



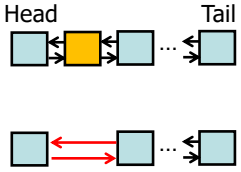
- LinkedList
  - Link-based impl of the List and Deque interfaces
  - Fast insertion and removal of elements



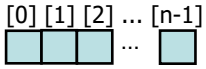
- ArrayList
  - Array-based impl of the List interface
  - Fast index-based access
  - Slow insertion and removal of non-tail elements
    - Fast insertion and removal of the tail element



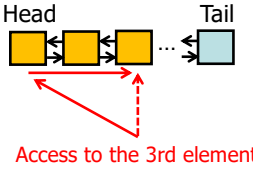
- LinkedList
  - Link-based impl of the List and Deque interfaces
  - Fast insertion and removal of elements



- 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.

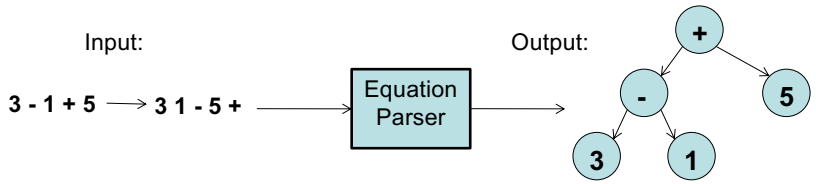


- Use ArrayList
  - If you often need to access "middle" elements with their index numbers
    - i.e., if index numbers mean something important/special.
- Use LinkedList
  - If you often need to insert/remove elements.
  - If you don't need index-based element access.
- Both yield the same performance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

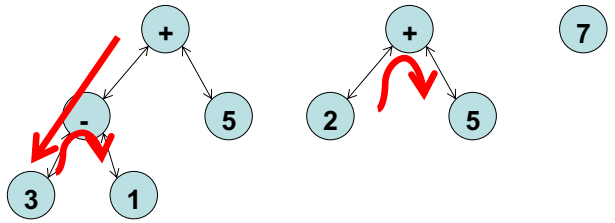
- ArrayList
  - Not that great, performance-wise, in multi-threaded programs.
    - Hard to make element traversal concurrent/parallel.
- LinkedList
  - More friendly for concurrency/parallerism.

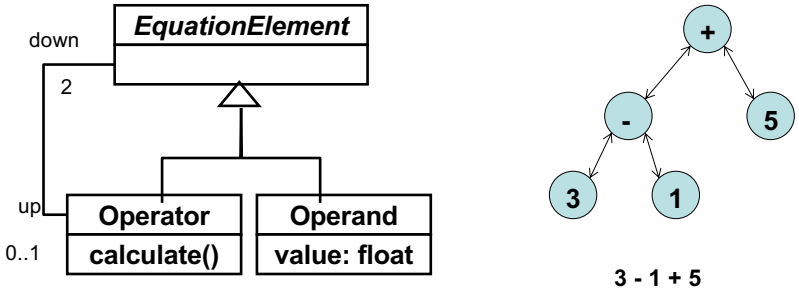
Another Example of *Composite*

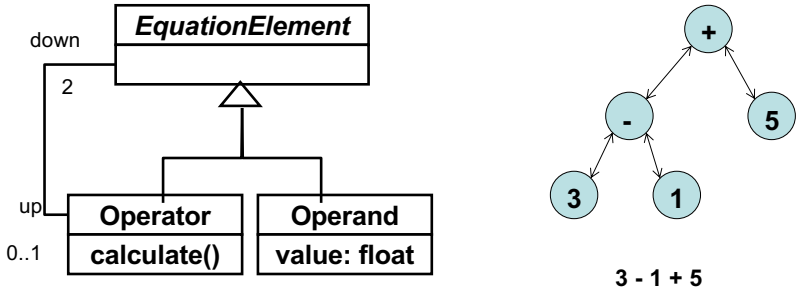
- Suppose you are developing a parser that parses an equation in a textual form.
  - Input: textual representation of an equation
  - Output: equivalent in-memory representation
    - Tree structure
      - Leaf nodes represent operands
      - The root and intermediate nodes represent operators.



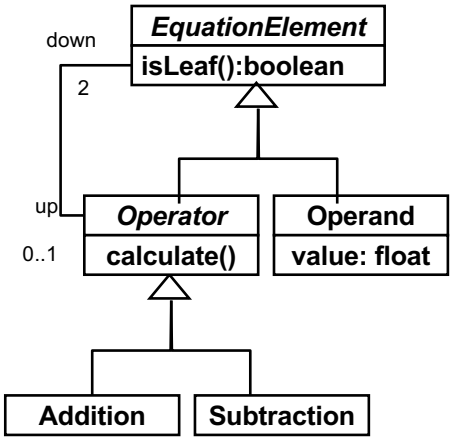
- This parser performs a depth-first element traversal.
  - Starts with the "deepest" and "left-most" leaf node
  - Traverse all nodes in the same layer
  - Goes up to a higher layer







- calculate() requires conditional branches to perform different types of arithmetic operations
  - e.g., +, -, etc.

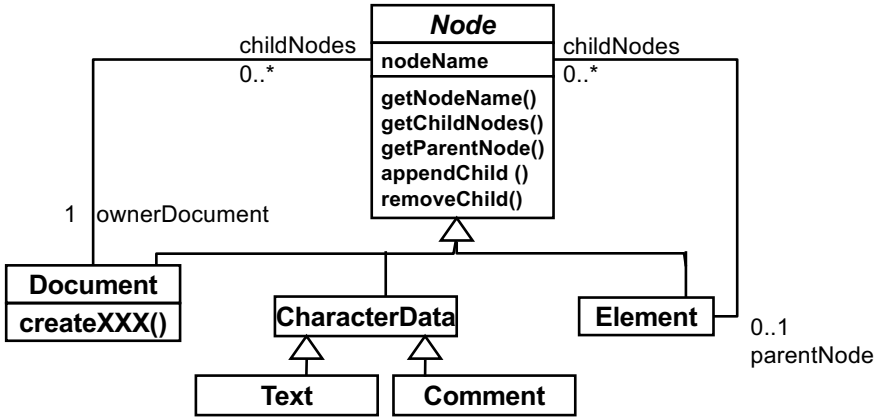


- Conditional branches can be eliminated in calculate() via polymorphism.

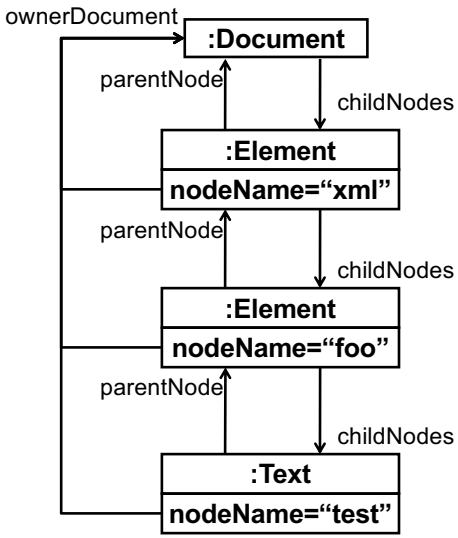
An Extra Example

- Document processing
  - Document Object Model (DOM)
    - A parser interface for XML parsers.
    - Specification: www.w3c.org

Simplified (Slightly Modified) DOM API



Example Instances of DOM Classes




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

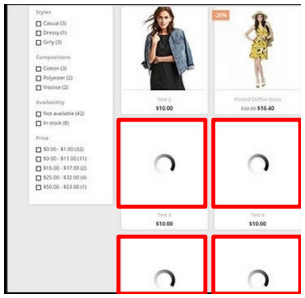

Proxy Design Pattern

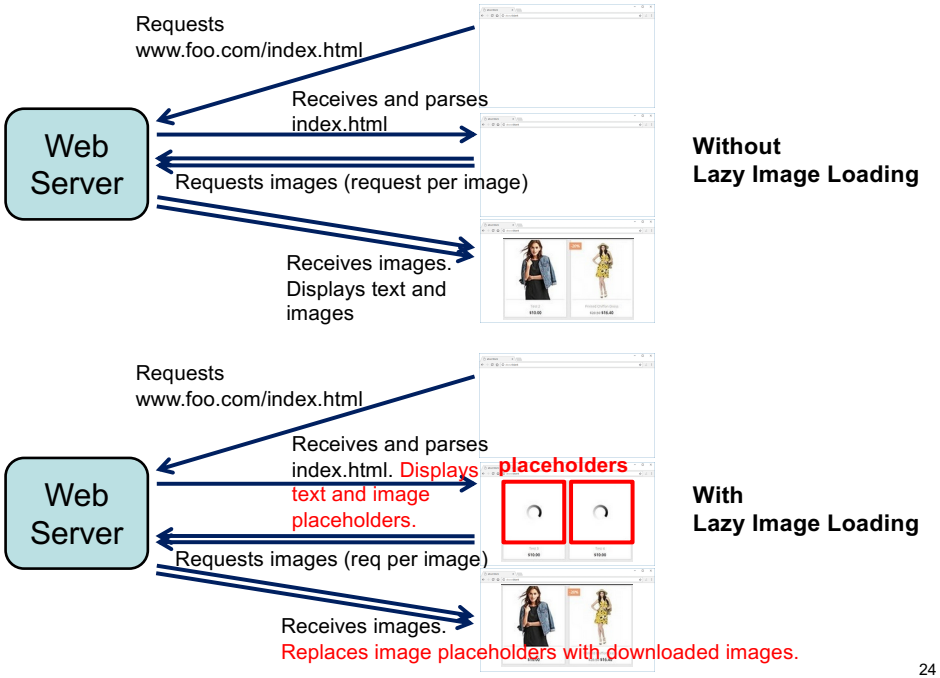
- Intent
  - Provide a surrogate or placeholder for another object to control access to it.

Proxy Design Pattern

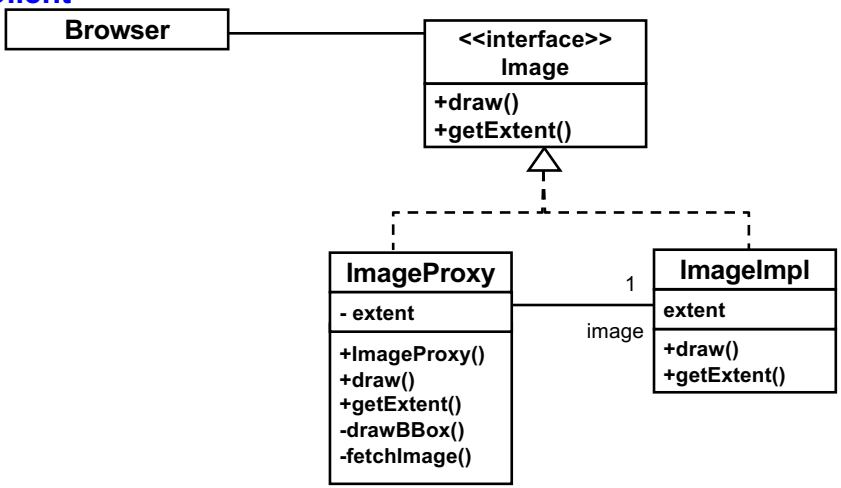
An Example: Lazy Image Loading in a Web Browser

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





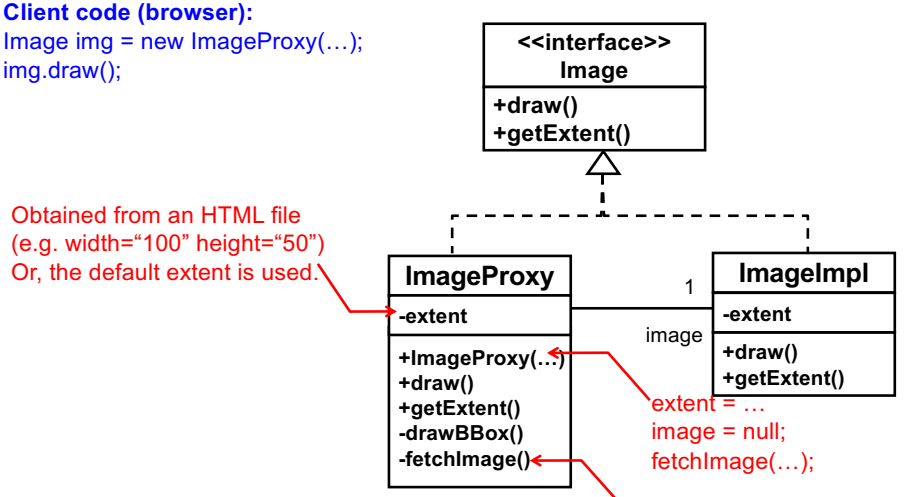
Client



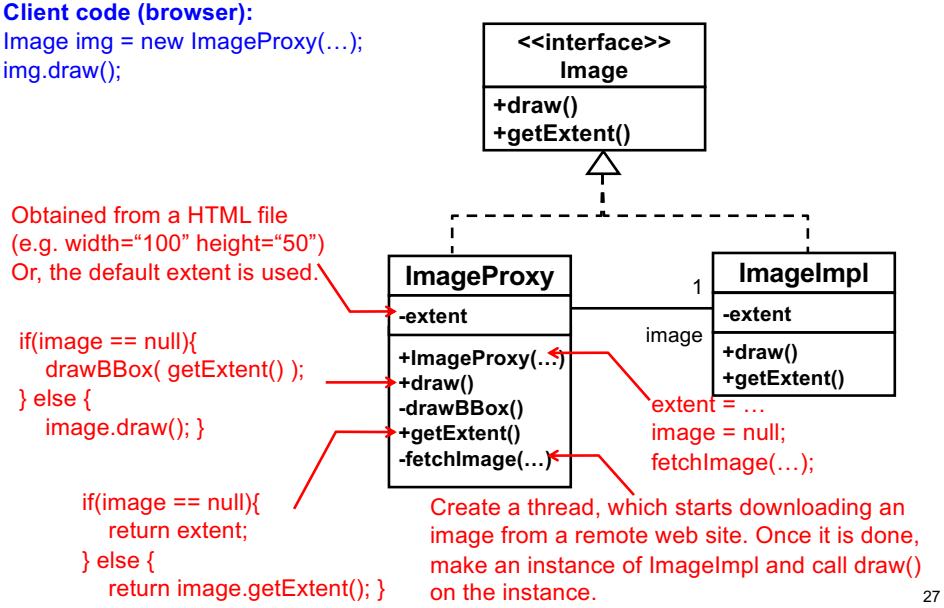
Proxy Class

Client code (browser):

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

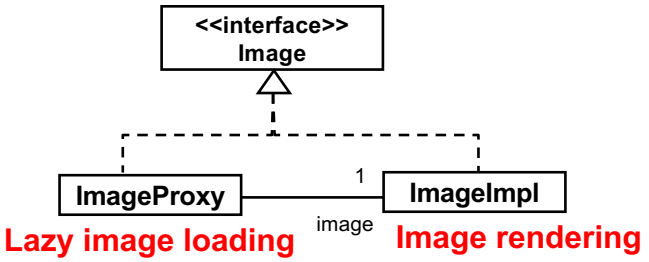


Client code (browser):
Image img = new ImageProxy(…);
img.draw();



What's the Point?

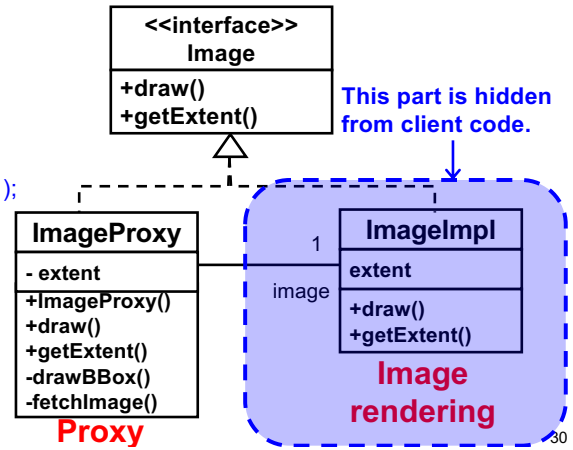
- Decouple (or loosely couple) *bounding box placement (lazy image loading)* and *image rendering*.
  - Make the two concerns independent with each other
    - Separation of concerns to improve maintainability



- Separation of concerns to improve 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 algorithm may be upgraded.

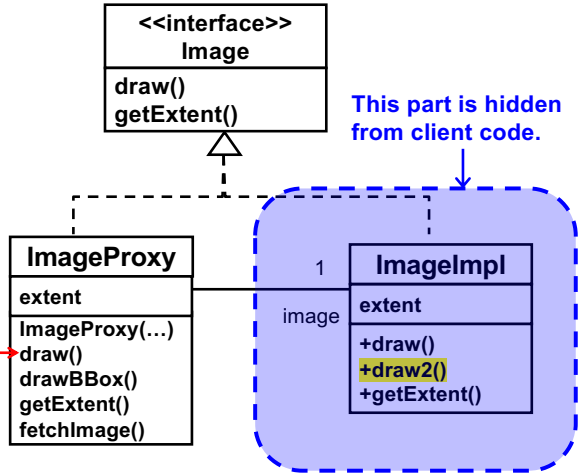
- *Proxy* can hide image rendering from its client.
  - The client (browser) uses (or faces) ImageProxy, not ImageImpl.
  - When a change is made on image rendering, you don't have to change client code.
    - New image formats may be introduced.
    - Rendering algorithm may be upgraded.

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



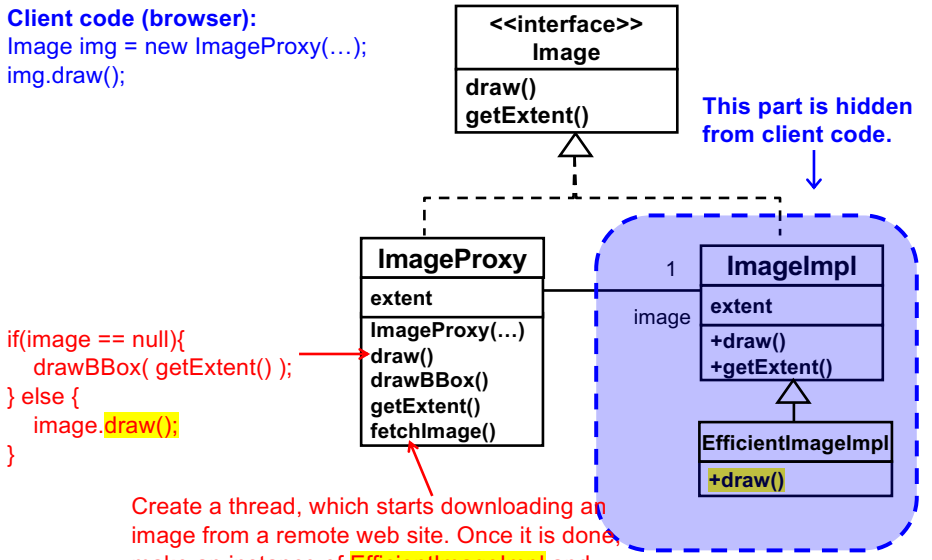
Supporting a New Rendering Algorithm

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




```
if(image == null){
   drawBBox( getExtent() );
} else {
   image.draw2();
//  image.draw();
}
```


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

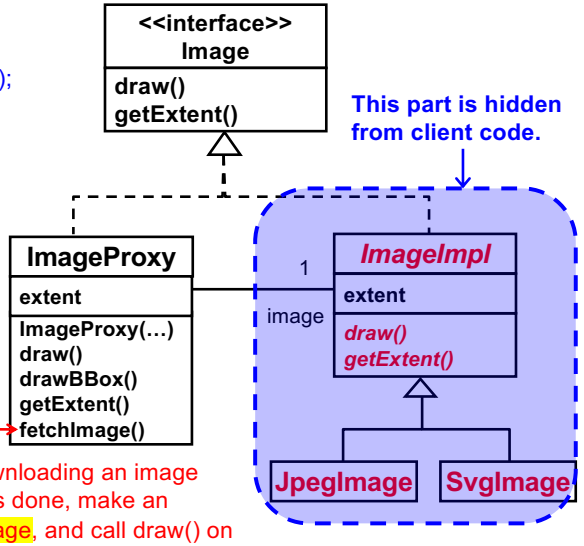



```
if(image == null){
   drawBBox( getExtent() );
} else {
   image.draw();
}
```

Create a thread, which starts downloading an image from a remote web site. Once it is done, make an instance of EfficientImageImpl and call draw() on the instance.

Supporting Multiple Image Formats

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

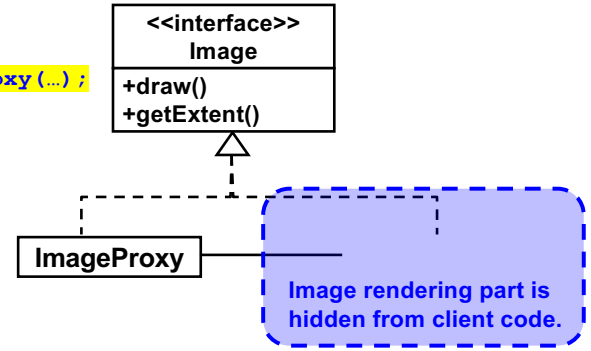


Create a thread, which starts downloading an image from a remote web site. Once it is done, make an instance of JpegImage or SvgImage, and call draw() on the instance.

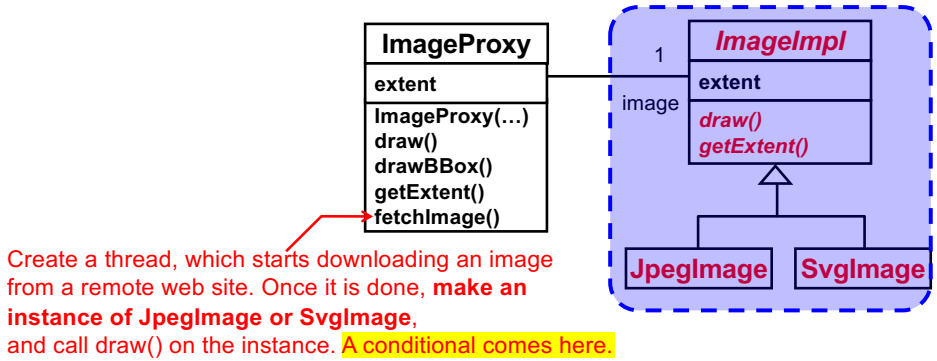
Two Possible Design Improvements

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 used).
    - Let's decouple `ImageProxy` and its client.

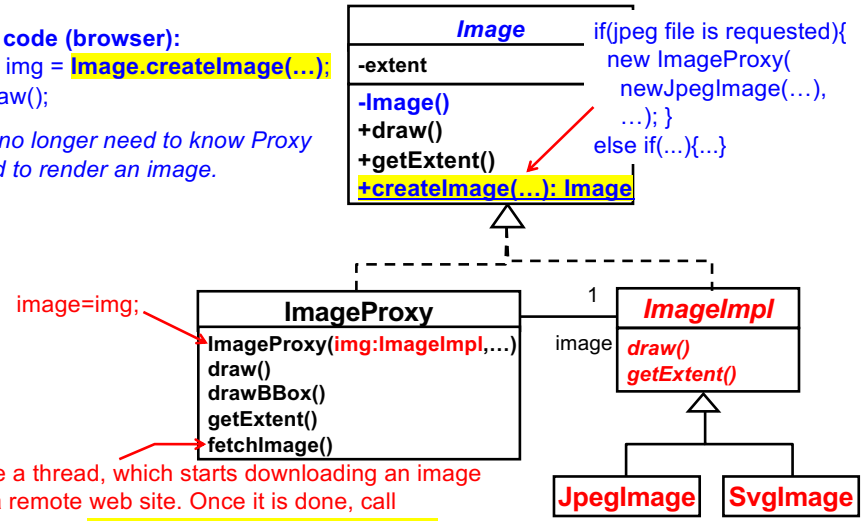


- `ImageProxy`
  - Now needs to know what image formats the browser supports.
  - Actually doesn't have to (want to) know that.
    - Let's decouple `ImageProxy` from the choice of image formats

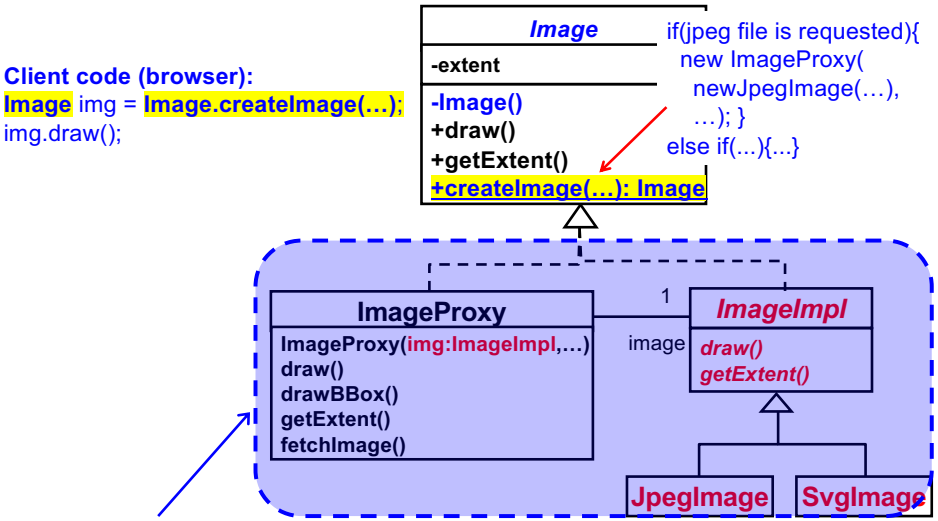
One Step Further with *Static Factory Method*

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

Client no longer need to know Proxy is used to render an image.



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



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

What if Everything is Integrated in a Single Class?

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

The choice of image formats, rendering algorithms and image loading are all mixed up in a single class, which will be fat and spaghetti (i.e. not that maintainable).

Better design strategy: Separation of concerns

Recap: Modularity

- Programming languages had no notion of structures (or modularity)
- As the size and complexity of software increased, languages needed modularity.
  - Module: A chunk of code
  - Modularity: Making modules self-contained and independent from others
  - Goal: Improve productivity and maintainability
    - Higher productivity through higher reusability
      - Lead to less production costs
    - Higher maintainability through clearer separation of concerns
      - Lead to less maintenance costs

Modules in SLs and OOPLs

- Modules in Structured Prog. Languages (SPLs)
  - Structure = a set of variables
  - Function = a block of code
- Modules in Object-Oriented PLs (OOPLs)
  - Class = a set of variables (data fields) and functions (methods)
  - Interface = a set of functions (methods)
- Key design questions/challenges:
  - how to define modules?
  - how to separate a module from others?
  - how to let modules interact with each other?