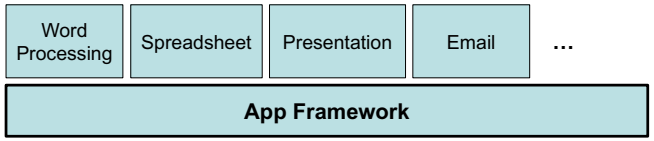
Factory Method

Factory Method

- A method to instantiate a class and initializes a class instance without using its constructors
  - Uses a regular (i.e., non-constructor) method.
  - Lets a class *defer* instantiation to subclasses.
    - Define an abstract class for creating an instance.
    - Let its subclasses decide *which class to instantiate*.

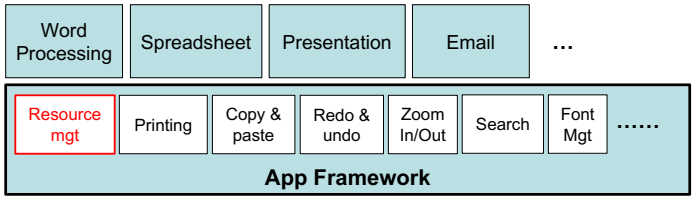
An Example: A Framework for Productivity ("Office") Applications

- Application framework
  - A set of foundation APIs to implement and run a series of applications.
    - Implement the standard/common functionalities (structures and behaviors) among individual applications
    - Make them reusable/available for individual apps.
    - Make app development easier and faster.
  - Frameworks for productivity ("office") applications
    - e.g., .Net Framework, Microsoft Foundation Class (MFC), Cocoa, OpenOffice Framework, GNOME, KDE, etc.

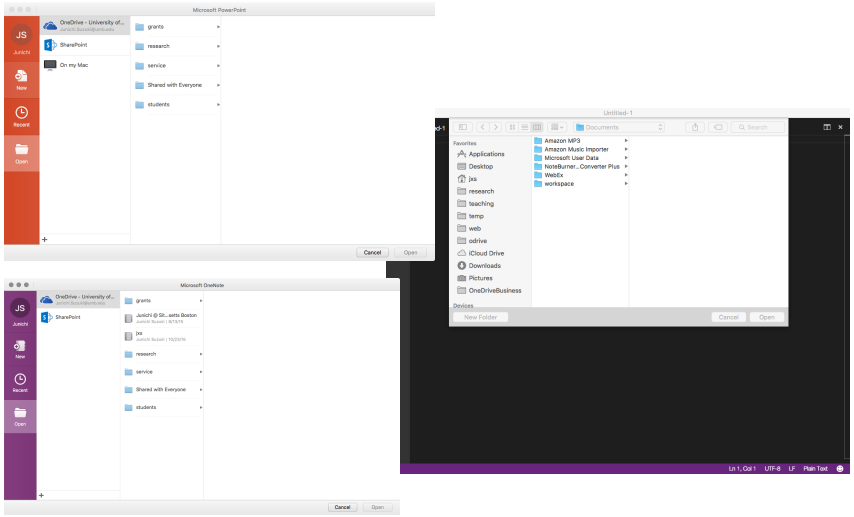


Resource Mgt in App Framework

- Resource management
  - Creating, opening and closing resources such as documents, spreadsheets, presentation sheets, emails and notes.
  - Saving resources in the local disk or a remote cloud.
  - Renaming resources.
  - Exporting resources as other formats.
- Here, we focus on the ***creation*** of resources.



In Microsoft Office Applications…

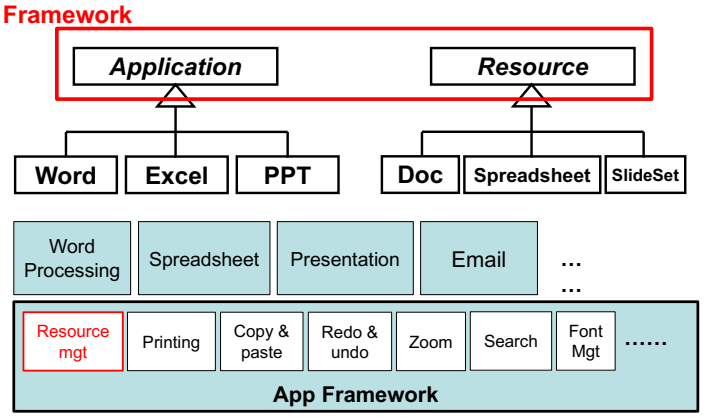


Assumptions for Resource Creation

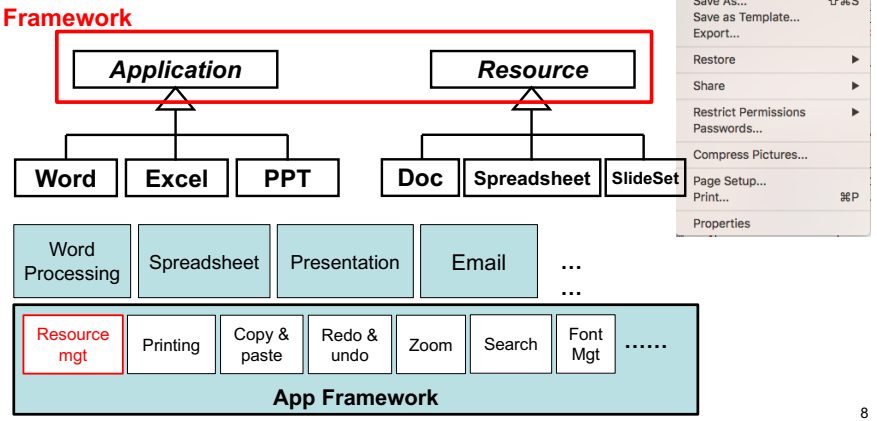
- Multiple applications exist on an app framework.
  - Extra applications may be developed in the near future.
  - Different applications create and use different types of resources.
  - When an application creates a new resource, it opens a blank resource.
  - Each application creates one resource at a time, but can keep multiple resources open.
  - Each application records the list of resources that it opened recently.

Requirements for Resource Creation

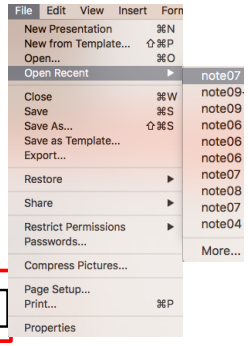
- Multiple applications exist.
- Different applications create and use different types of resources.

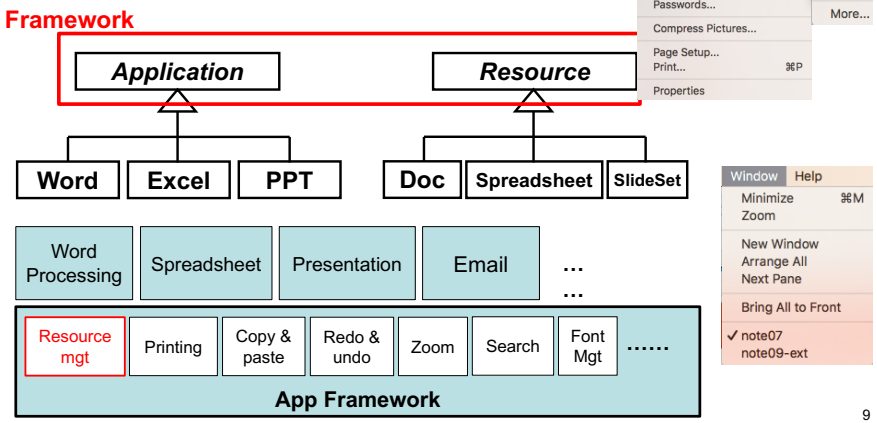


- When an application creates a new resource, it opens a blank resource.
  - Word should create a document.
  - Excel should create a spreadsheet.
  - PPT should create a slide set.

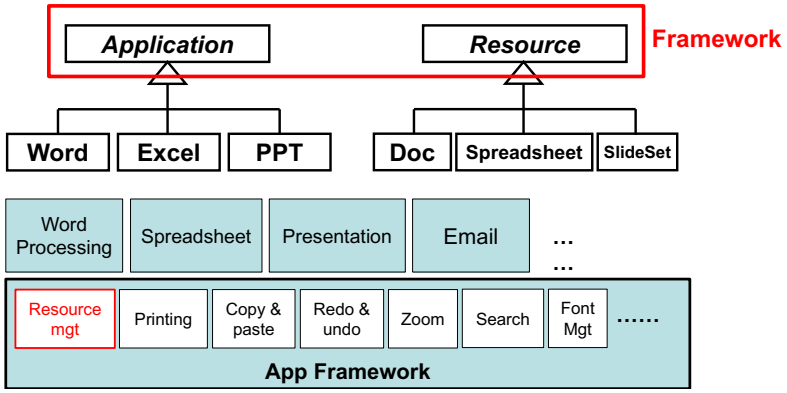


- Each application creates one resource at a time, but can keep multiple resources open.
- Each application records the list of resources that it opened recently.

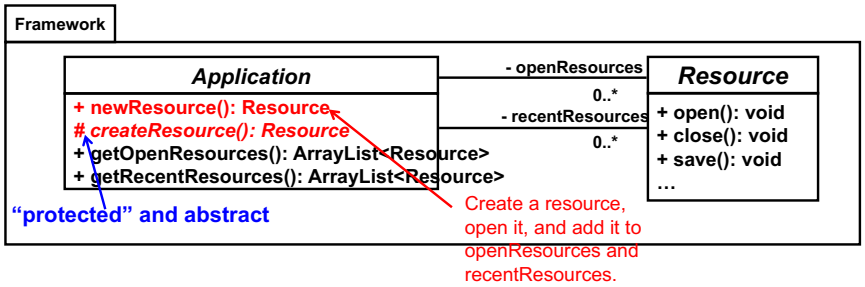


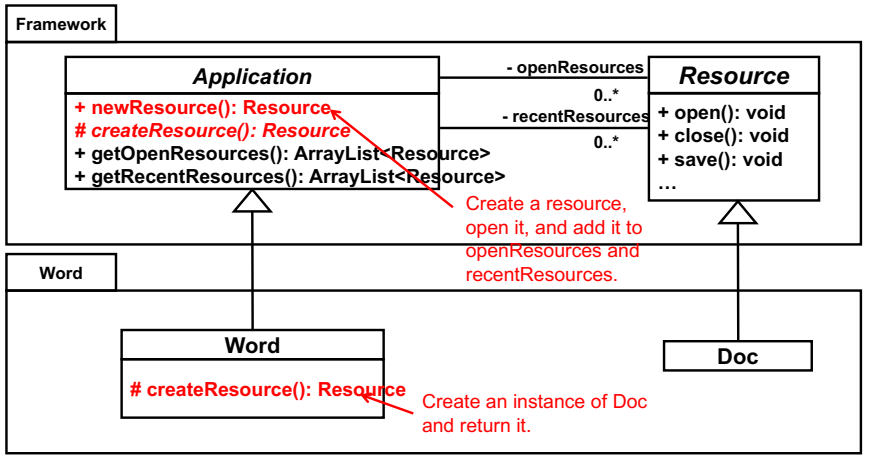


- Extra applications may be developed in the near future.
  - An app to be developed in the future should create a particular resource associated to that app.
    - We don't know the app-resource pair now.
- How can we implement the *common creation logic* in the framework level (i.e., with Application and Resource) without knowing Application's and Resource's subclasses?

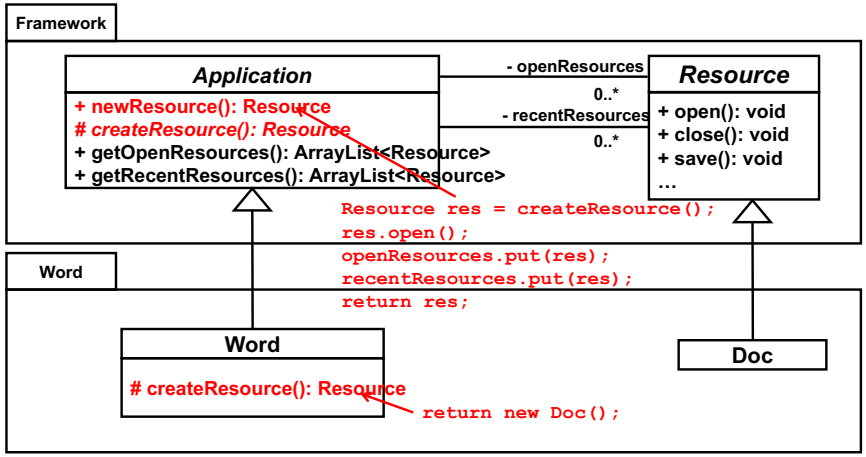


Solve this Design Issue with *Factory Method*






```
Word word = new Word(…);
word.newResource();
```




```
Word word = new Word(…);
word.newResource();
```


What's the Point?

- The framework
  - `newResource()` provides a *skeleton* (or *template*) for resource creation.
    - *Partially* implements a common procedural sequence for resource creation.
  - Never specify specific types (specific class names) for apps and their resources.
- Word (framework client)
  - Reuses the skeleton/template for resource creation and *completes* it
    - By specifying which application class and which resource class are used.

What *Factory Method* Does…

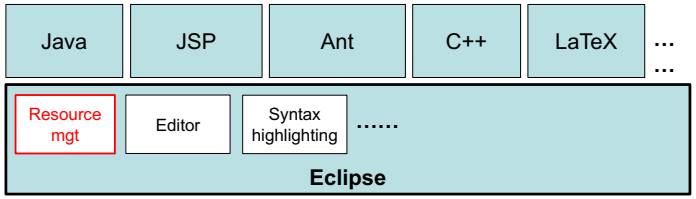
- Define a *factory method* (`newResource()`) in `Application`.
- Have it implement a common procedural sequence (a skeleton or template) for resource creation and initialization with an *empty protected method* (`createResource()`).

Benefits

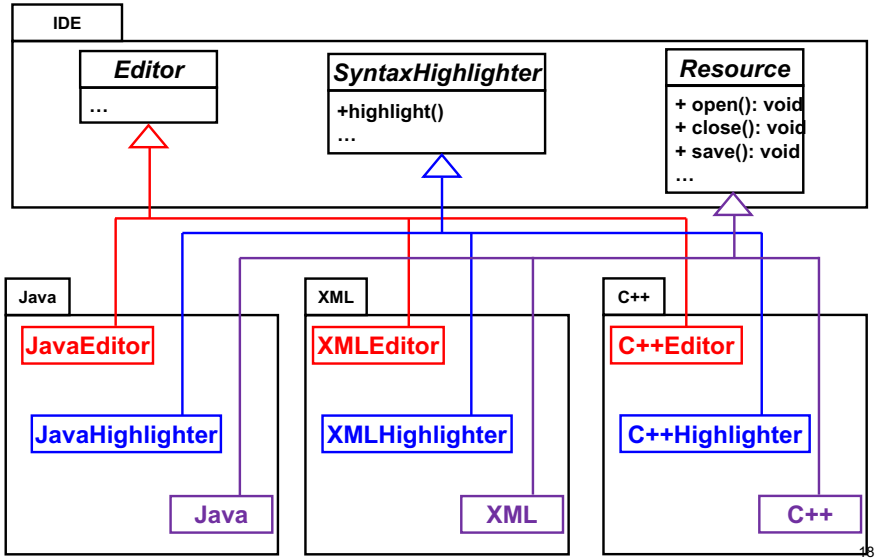
- The framework
  - Can define a common procedural sequence (a skeleton or template) for resource creation and initialization.
  - Allows individual apps to reuse it.
    - Less redundant code in apps.
  - Does not have to know app-resource pairs (i.e., which specific apps uses which specific resources).
  - Can "force" every single app to follow the same behavior (i.e. same sequence for instance creation and initialization) when it creates a new resource.

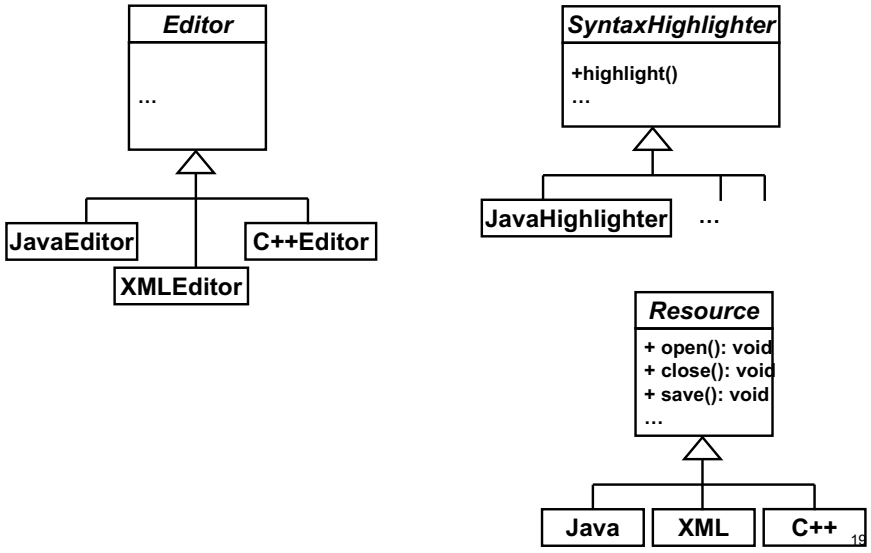
Another Example: Resource Mgt in Integrated Development Environments (IDEs)

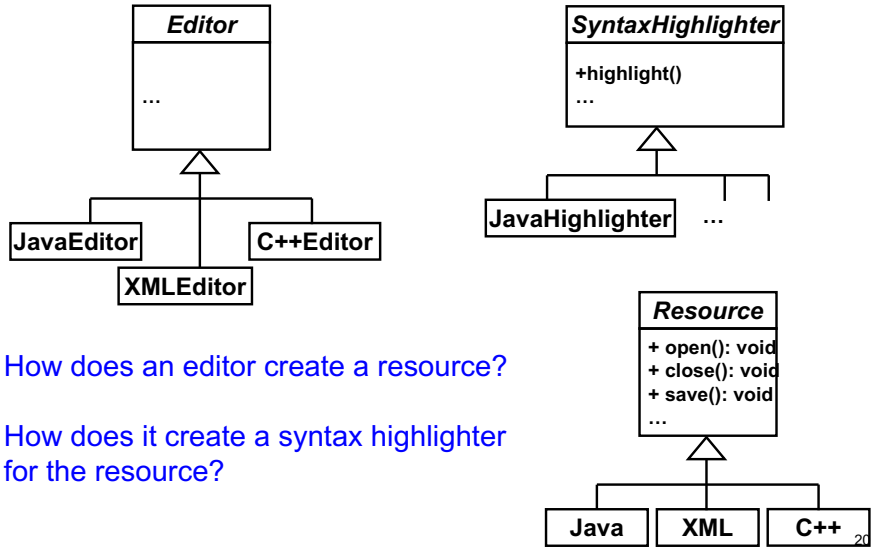
- Imagine an IDE like Eclipse, for example.
- Resources in an IDE
  - Programs (e.g., Java, JavaScript, C++, etc.)
  - XML files (e.g., build.xml for Ant, web.xml for Servlet WAR)
  - …, etc.
- Many IDE components (e.g., plugins) use resources.
  - Editors, syntax highlighters, etc.



Editors, Syntax Highlighters and Resources



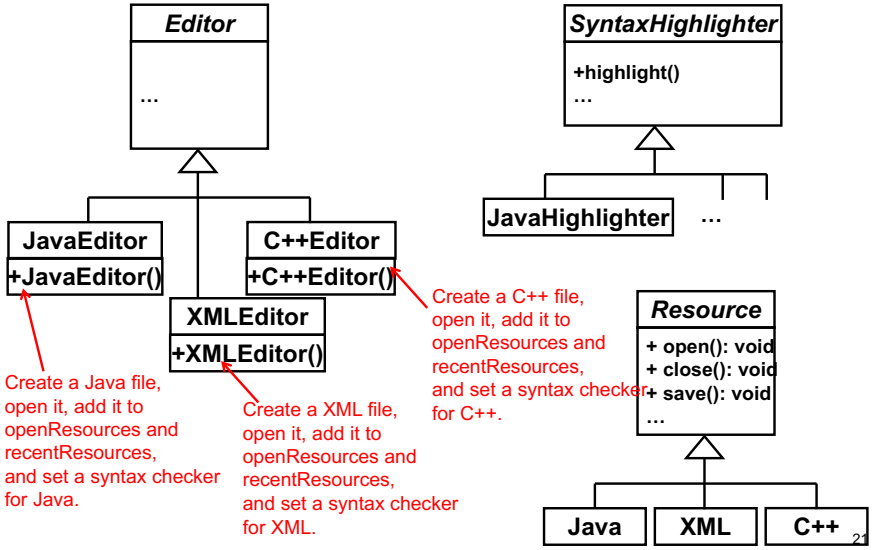




How does an editor create a resource?

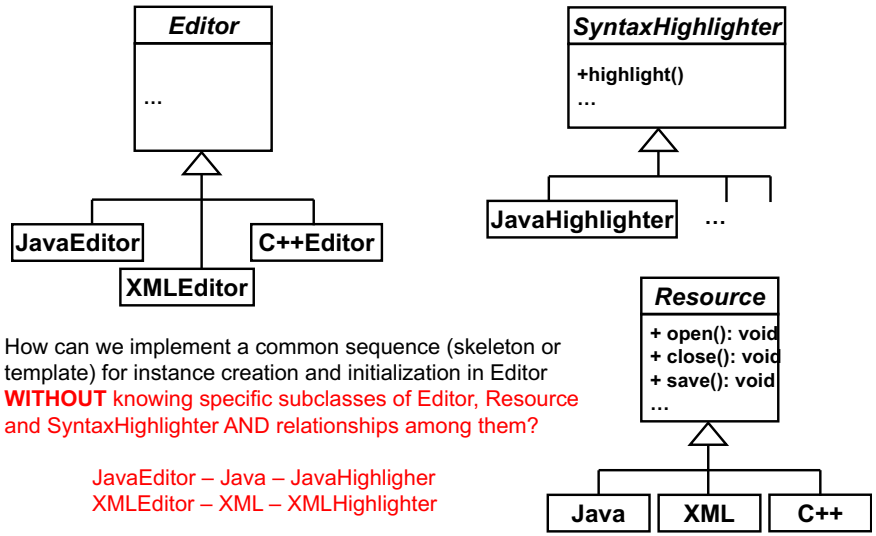
How does it create a syntax highlighter for the resource?

If We don't *Factory Method…*



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

Dilemma

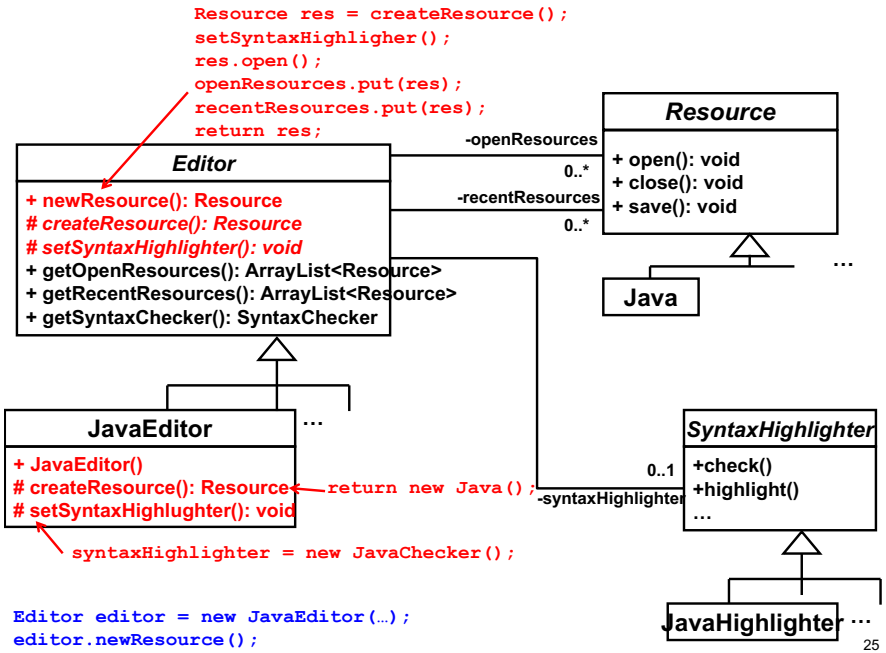


How can we implement a common sequence (skeleton or template) for instance creation and initialization in Editor **WITHOUT** knowing specific subclasses of Editor, Resource and SyntaxHighlighter AND relationships among them?

JavaEditor – Java – JavaHighligher
XMLEditor – XML – XMLHighlighter

What to Do with *Factory Method*

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

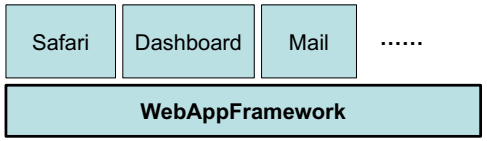


Benefits

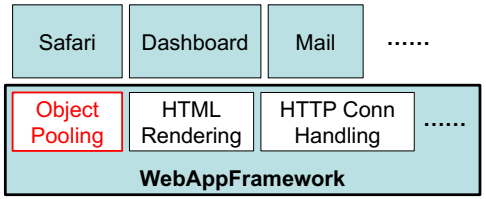
- This IDE
  - Can define a common sequence (a skeleton or template) for instance creation and initialization
  - Allows individual editors to reuse it.
    - Less redundant code
  - Does not have to know editor-resource-syntax highlighter mappings.
  - Can "force" every single editor to follow the same behavior (i.e. same sequence for instance creation and initialization) when it creates a new resource.

One More Example: Web App Dev Framework

- Assume you are implementing a development framework for various web apps.
  - Frameworks for web applications
    - e.g., WebKit, Struts, Ruby on Rails, etc.
    - WebKit (http://www.webkit.org/)
      - Web browsers (incl. Safari), Dashboard, Mail and many other Mac OS X apps.



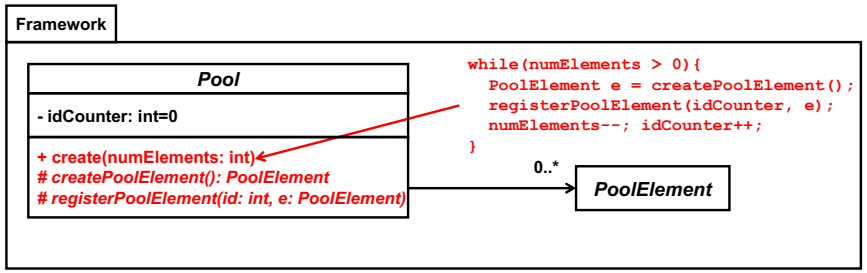
- Assume you are implementing an object pooling API in this framework.
  - Creates and manages a pool of (the same kind/type of) objects
    - e.g., a pool of browser windows, a pool of tabs in each browser window, a pool (cache) of HTML files, a pool of HTTP connections, a pool of threads, etc.
- Here, we focus on the *creation* of pools.



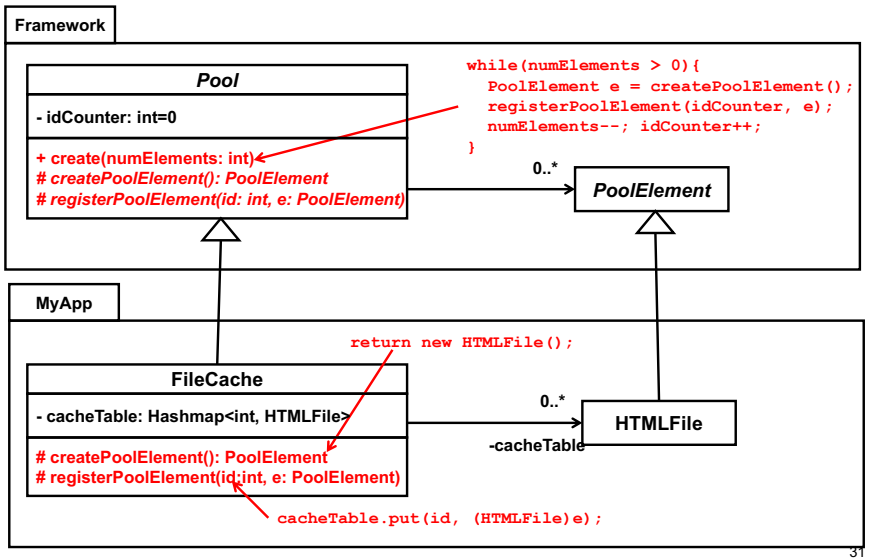
Background: Web Page Caching

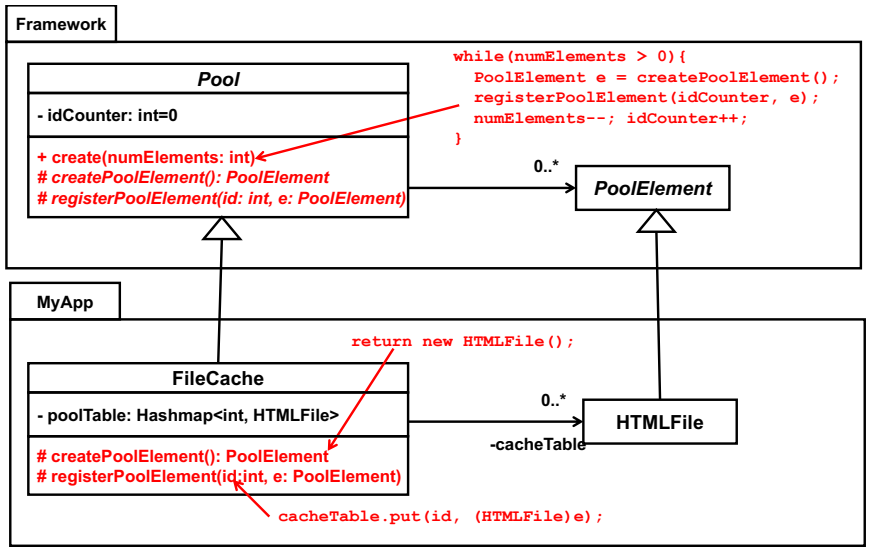
- A web server
  - Receives a connection establishment request from a client (browser).
  - Receives an HTTP command
  - Retrieves a target HTML file from the local disk and returns it to the client.
  - May cache a set of HTML files that have been accessed in the past.
    - Keep them in the memory
    - Faster response to the clients
      - Memory access is much faster than disk access.

Factory Method in Object Pooling



HTML File Caching

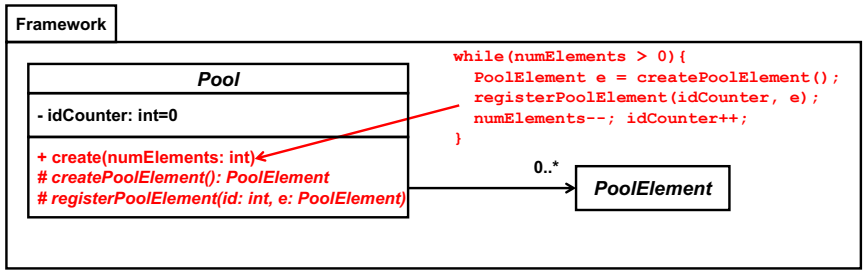




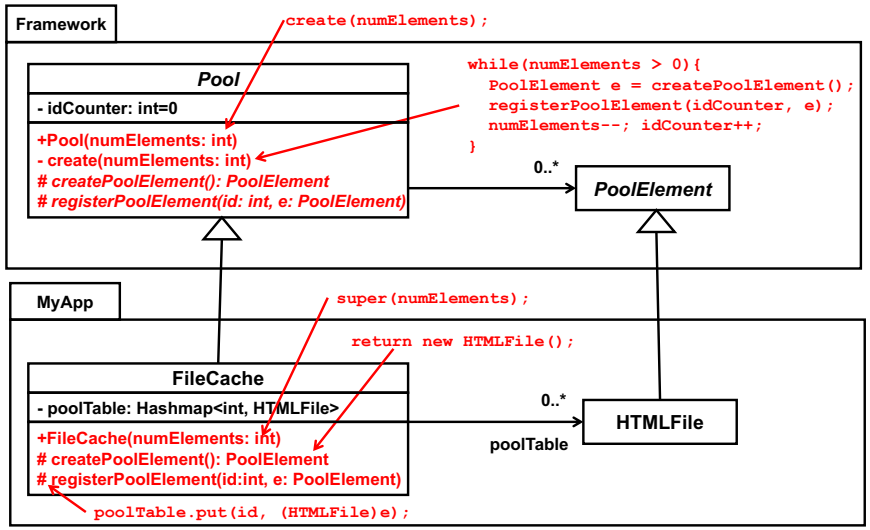

```
FileCache cache = new FileCache();
cache.create(10);
```

Factory Method

- createPoolElement()
  - Allows create() to avoid stating the class name for pool elements.
  - Allows the framework to be independent (or de-coupled) from individual applications (framework clients).
    - Allows applications to be pluggable to the framework.



One Step Further



Client code gets simpler: `FileCache cache = new FileCache(10);`

Static Factory Method and *Factory Method*

- *Static factory method* is a variant (or a special case) of *Factory Method*.