May
20-05-2025
Tuesday

Day - 8
Lecture - 7

Build Google Docs | Document Editor LLD

Problem Statement: We have to make a document editor where we can edit text and image for now.

Scalable
- tables
- video
- fonts
- new line
- tab space

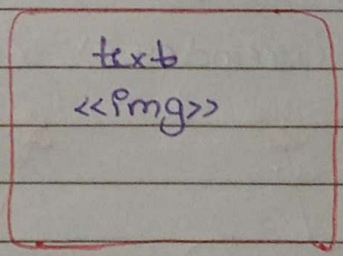
We can add this but we focus only on text and img

Two ways to approach LLD Problem.

- Top-down Approach: first we main objects of the application.
- Bottom-up Approach: First we make small-small objects and their dependencies.

Preference to use Bottom-up some problem require top-down Approach.

Document Editor

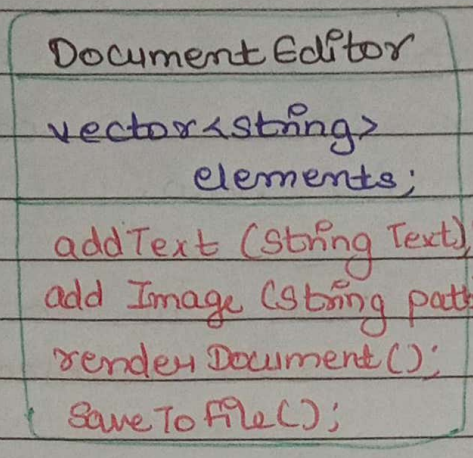


* Bad Design
→ We make Document Editor class
→ For text and img if we use different list then it will become messy so we use only list for both.
→ text and img will be both store as string and we give the path of img
→ Method addText (string Text) addImg (string Path)
renderDocument (); → elements ko loop through karega
→ saveToFile ();

```cpp
// Bad Design

#include <iostream>
#include <vector>
#include <string>
#include <fstream>

using namespace std;

class DocumentEditor {
private:
    vector<string> documentElements;
    string renderedDocument;

public:
    // Adds text as a plain string
    void addText(string text) {
        documentElements.push_back(text);
    }

    // Adds an image represented by its file path
    void addImage(string imagePath) {
        documentElements.push_back(imagePath);
    }

```

```
    // Renders the document by checking the type of each element at runtime
    string renderDocument() {
        if(renderedDocument.empty()) {
            string result;
            for (auto element : documentElements) {
                if (element.size() > 4 && (element.substr(element.size() - 4) == ".jpg" ||
                 element.substr(element.size() - 4) == ".png")) {
                    result += "[Image: " + element + "]" + "\n";
                } else {
                    result += element + "\n";
                }
            }
            renderedDocument = result;
        }
        return renderedDocument;
    }

    void saveToFile() {
        ofstream file("document.txt");
        if (file.is_open()) {
            file << renderDocument();
            file.close();
            cout << "Document saved to document.txt" << endl;
        } else {
            cout << "Error: Unable to open file for writing." << endl;
        }
    }
};
```

```
int main() {
    DocumentEditor editor;
    editor.addText("Hello, world!");
    editor.addImage("picture.jpg");
    editor.addText("This is a document editor.");

    cout << editor.renderDocument() << endl;

    editor.saveToFile();

    return 0;
}

```

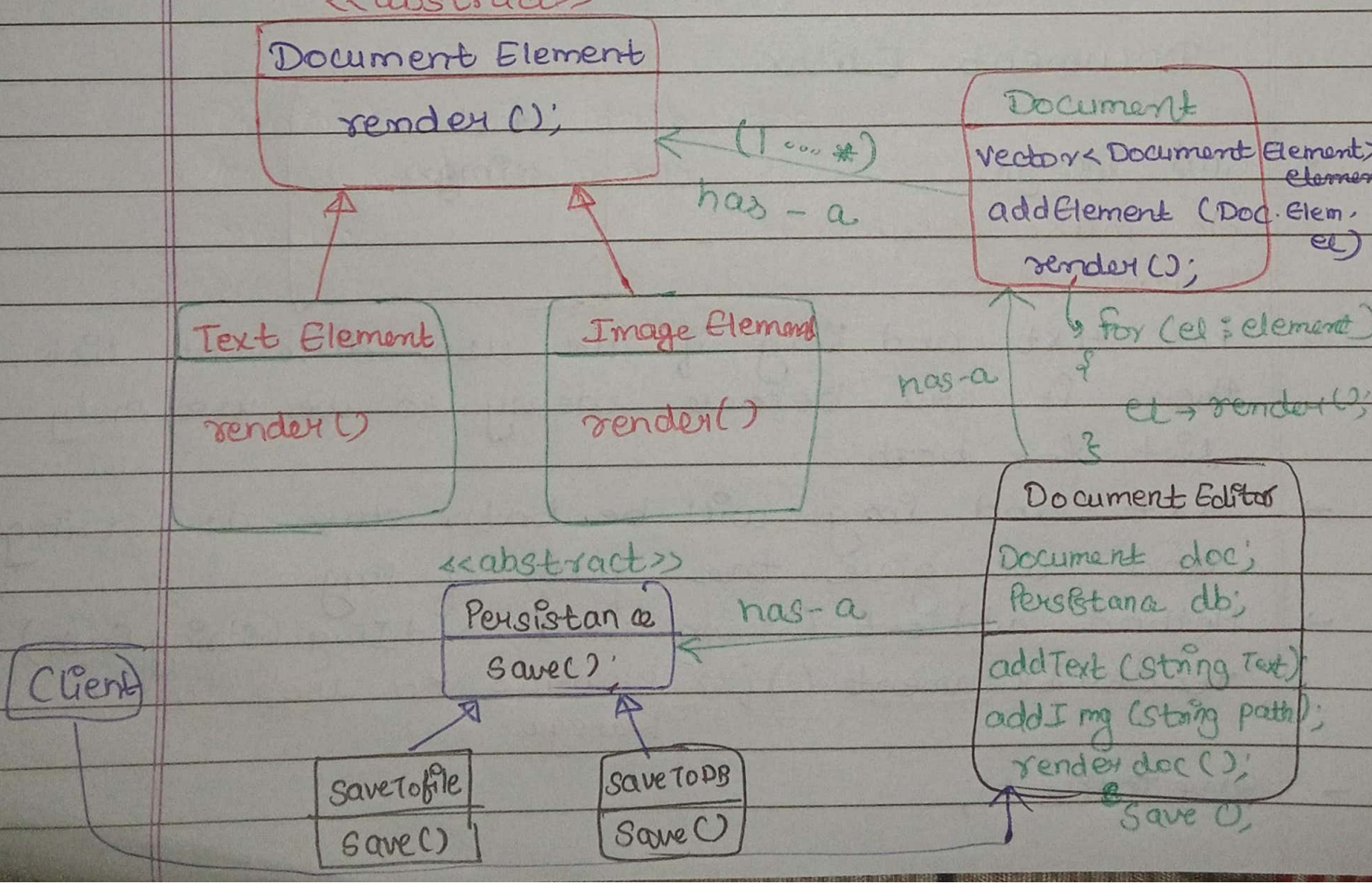


Problems with Bad Design

→ Breaking SRP (Document editor do all things; add text, img)

→ Breaking OCP (we can't add new features because we have to make changes in one class)

Better Design



```cpp
// Good Design

#include <iostream>
#include <vector>
#include <string>
#include<fstream>

using namespace std;

// Abstraction for document elements
class DocumentElement {
public:
    virtual string render() = 0;
};

// Concrete implementation for text elements
class TextElement : public DocumentElement {
private:
    string text;

public:
    TextElement(string text) {
        this->text = text;
    }

    string render() override {
        return text;
    }
};

```

```
// Concrete implementation for image elements
class ImageElement : public DocumentElement {
private:
    string imagePath;

public:
    ImageElement(string imagePath) {
        this->imagePath = imagePath;
    }

    string render() override {
        return "[Image: " + imagePath + "]";
    }
};

// NewLineElement represents a line break in the document.
class NewLineElement : public DocumentElement {
public:
    string render() override {
        return "\n";
    }
};

// TabSpaceElement represents a tab space in the document.
class TabSpaceElement : public DocumentElement {
public:
    string render() override {
        return "\t";
    }
};

```

```cpp
// Document class responsible for holding a collection of elements
class Document {
private:
    vector<DocumentElement*> documentElements;

public:
    void addElement(DocumentElement* element) {
        documentElements.push_back(element);
    }

    // Renders the document by concatenating the render output of all elements.
    string render() {
        string result;
        for (auto element : documentElements) {
            result += element->render();
        }
        return result;
    }
};

// Persistence abstraction
class Persistence {
public:
    virtual void save(string data) = 0;
};

// FileStorage implementation of Persistence
class FileStorage : public Persistence {
public:
    void save(string data) override {
        ofstream outFile("document.txt");
        if (outFile) {
            outFile << data;
            outFile.close();
            cout << "Document saved to document.txt" << endl;
        } else {
            cout << "Error: Unable to open file for writing." << endl;
        }
    }
};

```

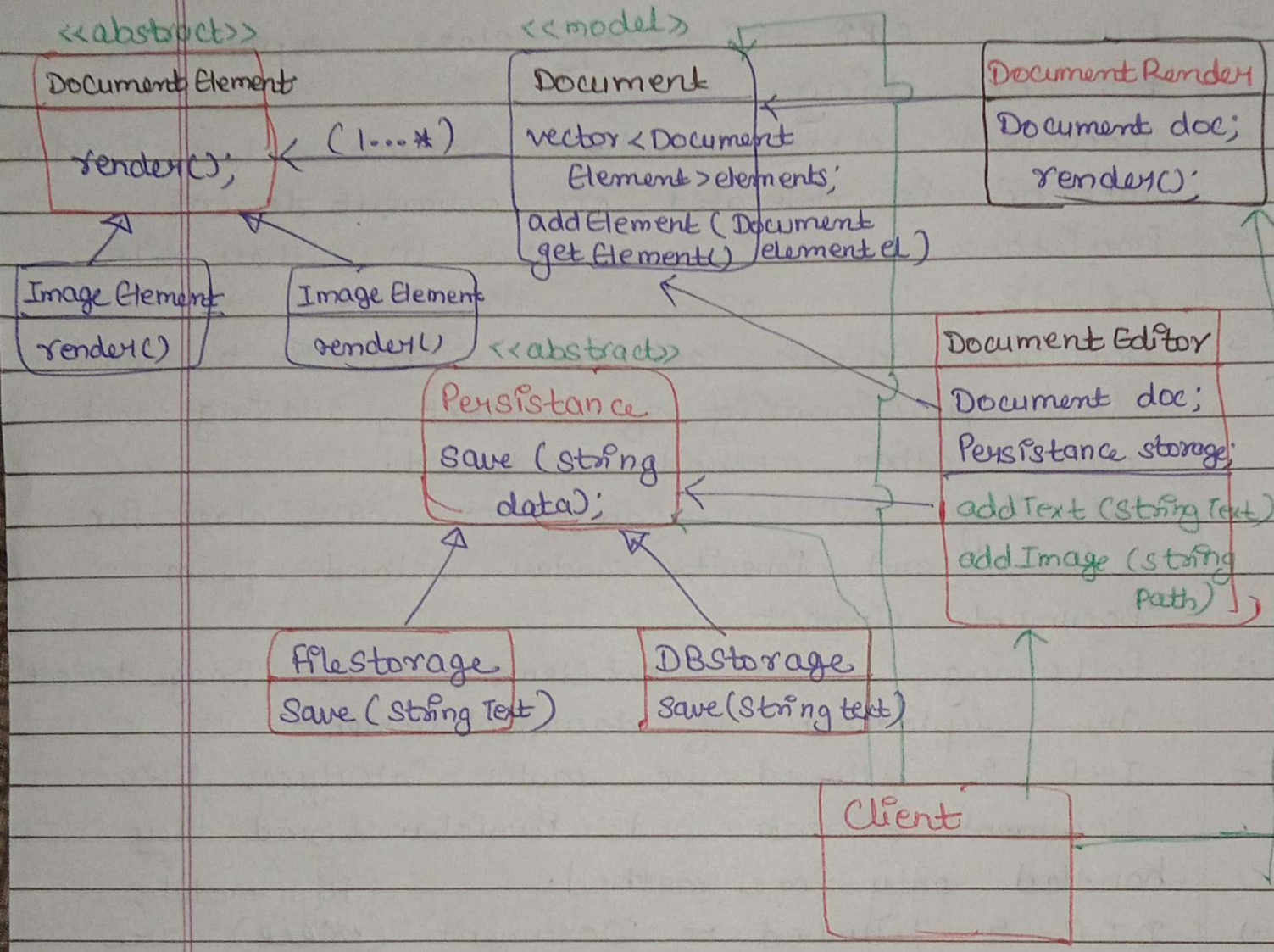
```
// Placeholder DBStorage implementation
class DBStorage : public Persistence {
public:
    void save(string data) override {
        // Save to DB
    }
};

// DocumentEditor class managing client interactions
class DocumentEditor {
private:
    Document* document;
    Persistence* storage;
    string renderedDocument;

public:
    DocumentEditor(Document* document, Persistence* storage) {
        this->document = document;
        this->storage = storage;
    }

    void addText(string text) {
        document->addElement(new TextElement(text));
    }

    void addImage(string imagePath) {
        document->addElement(new ImageElement(imagePath));
    }

    // Adds a new line to the document.
    void addNewLine() {
        document->addElement(new NewLineElement());
    }

    // Adds a tab space to the document.
    void addTabSpace() {
        document->addElement(new TabSpaceElement());
    }
```

```
141
142     string renderDocument() {
143         if(renderedDocument.empty()) {
144             renderedDocument = document->render();
145         }
146         return renderedDocument;
147     }
148
149     void saveDocument() {
150         storage->save(renderDocument());
151     }
152 };
153
154 // Client usage example
155 int main() {
156     Document* document = new Document();
157     Persistence* persistence = new FileStorage();
158
159     DocumentEditor* editor = new DocumentEditor(document, persistence);
160
161     // Simulate a client using the editor with common text formatting features.
162     editor->addText("Hello, world!");
163     editor->addNewLine();
164     editor->addText("This is a real-world document editor example.");
165     editor->addNewLine();
166     editor->addTabSpace();
167     editor->addText("Indented text after a tab space.");
168     editor->addNewLine();
169     editor->addImage("picture.jpg");
170
171     // Render and display the final document.
172     cout << editor->renderDocument() << endl;
173
174     editor->saveDocument();
175
176     return 0;
177 }
178
```

→ following SRP – Every class responsible for one thing.
→ Document class handles element vector. Render method is delegated on document element.
→ Persistance class save data to either file or dB
→ Document Editor class talk with client and take functionality of add Text, add Img and make other methods delegated.
→ following OCP we can make class for fonts and inherits render method from Document element
→ following LSP – Text Element and Image element are replaceable of document element
→ ISP is followed we make interface like Document Element and Persistance and they handled only one method.
→ DIP is followed – Document (client) are not directly interact with Text Element (low-module). There is interface Document Element (High-module)

* Enhancement in Our Design

→ There is only thin line difference whether the principle are followed or not.
→ In Document class there is a element list so it should perform crud only on document element but it also handle render() but actually it's not rendering it delegate to Document Element Class.
→ We make different class for render();



Principle of least knowledge
You should always talk to your immediate friend.