2020-2021 Güz Dönemi

EGE ÜNİVERSİTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ
NESNEYE DAYALI PROGRAMLAMA
PROJE 2
NOTEPAD with Design Patterns

Sıla Eryılmaz 05180000002

Melek Şimal Ünsal 05180000110

Oksana Damarja 05170000811

Nesneye Dayalı Programlama Proje 2
Sıla Eryılmaz - 05180000002
Melek Şimal Ünsal - 05180000110
Oksana Damarja - 05170000811

İÇİNDEKİLER

Nesneye Dayalı Programlama Proje 2
Sıla Eryılmaz - 05180000002
Melek Şimal Ünsal - 05180000110
Oksana Damarja - 05170000811

PROJEDE KULLANILAN TASARIM DESENLERİ VE AÇIKLAMALARI

1. Command Tasarım Deseni

Bir nesne üzerinde bir işleminin nasıl yapıldığını bilmediğimiz ya da kullanılmak istenen nesneyi tanımadığımız durumlarda, Command tasarım deseni ile yapılmak istenen işlemi bir nesneye dönüştürerek, alıcı nesne tarafından işlemin yerine getirilmesi sağlayabiliyoruz.

Bu tasarım desenini yazılan bir karakteri geri almak için kullandık.

→ İlk olarak Command.java class'ını oluşturduk. Bu class bir interface'dir ve içinde void tipinde execute() bulundurur.

```
public interface Command {

    public void execute();

}
```

→ Daha sonra UndoableCommand.java class'ını oluşturuyoruz. Bu class bir interface'dir ve Command'den extend edilir. İçinde undo adı verilen ve parametre olarak UndoableEditEvent tipinde obje alan bir fonksiyon içerir.

```
public interface UndoableCommand extends Command {

    public void undo(UndoableEditEvent undoableEditEvent);

}
```

→ Notepad.java main class'ımızda undo için gerekli objeleri oluşturuyoruz.

```
//for undo and event
UndoManager undoManager = new UndoManager();
UndoableEditEvent editEvent; //created to detect and save changes made to the document
UndoableCommand undoableCommand = new UndoLetter(undoManager, editEvent);
```

→ UndoableCommand tipindeki nesnemiz ile undo işlemini çağırıp yapılan tüm işlemleri kaydediyoruz.

```
doc.addUndoableEditListener(new UndoableEditListener() {
    public void undoableEditHappened(UndoableEditEvent evt) {
        undoableCommand.undo(evt);
        // belgedeki her işlem kaydedilir.
    }
});
```

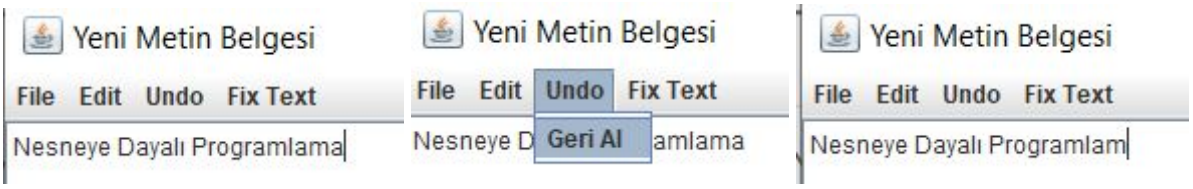
→Undo butonuna tıklandığında bir karakterin silinmesi için execute fonksiyonumuzu çağırıyoruz.

```
//Undo Button Action
Action Undo = new AbstractAction("Geri Al") {
    @Override
    public void actionPerformed(ActionEvent e) {
        undoableCommand.execute();
    }
};
```

→ UndoLetter.java class'ı ise UndoableCommand'den extend edilir ve içinde hem undo hem execute fonksiyonlarını bulundurur. undo() fonksiyonu ile yapılan tüm işlemler kaydedilir. execute() fonksiyonu ile geri alma işlemi gerçekleştirir.

```
public class UndoLetter implements UndoableCommand {

    public UndoManager manager;
    UndoableEditEvent editEvent;

    public UndoLetter(UndoManager manager, UndoableEditEvent editEvent) {
        this.manager = manager;
        this.editEvent = editEvent;
    }

    @Override
    public void undo(UndoableEditEvent undoableEditEvent) {
        manager.addEdit(undoableEditEvent.getEdit());
    }

    @Override
    public void execute() {
        if (manager.canUndo()) {
            manager.undo(); //undo letter
        } else {
            //When no more letter in textfile
            infoBox("No more letters!", "Warning!");
        }
    }

    public static void infoBox(String infoMessage, String titleBar) {
        JOptionPane.showMessageDialog(null, infoMessage, "InfoBox: " + titleBar, JOptionPane.INFORMATION_MESSAGE);
    }

}
```

Kullanıcı Arayüzünde Uygulaması:



2. Iterator Tasarım Deseni (Iterator Design Pattern)

Iterator design pattern, bir listenin yapısının ve çalışma tarzının uygulamanın diğer kısımları ile olan bağlantılarını en aza indirmek için; listede yer alan nesnelerin, sırasıyla uygulamadan soyutlanması amacıyla kullanılır.

Bu tasarım desenini “Single Transposition” butonuna basıldığında gerçekleşebilecek durumlar için kullandık. Iterator deseni için Java’nın

Nesneye Dayalı Programlama Proje 2
Sıla Eryılmaz - 05180000002
Melek Şimal Ünsal - 05180000110
Oksana Damarja - 05170000811

hazır paketleri import java.util.Iterator, import java.util.ListIterator şeklinde import edildi.

→ SingleTransposition.java class'ını inceleyecek olursak checkASCIIValues fonksiyonunda daha önce for döngüsü ile kontrol ettiğimiz wordsArray ve textPaneArray listesi tasarım deseni ile birlikte

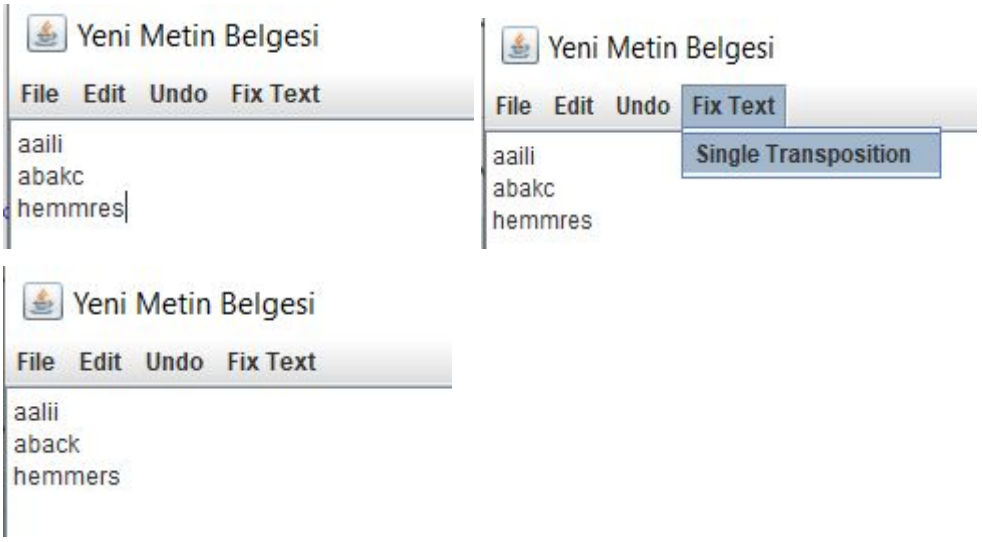
```
public void checkASCIIValues(JTextPane textPane) throws BadLocationException {

    /*When searching for single transpositions, we first find the ones whose ascii values are equal to each other,
     and we are eliminating to make fewer comparisons rather than the whole list.*/
    iterator = wordsArray.iterator();
    iteratorTextPaneArray = textPaneArray.iterator();
    while (iteratorTextPaneArray.hasNext()){
        sameASCIIarr = new ArrayList<>();
        String textArrayElement = iteratorTextPaneArray.next().toString();
        iterator = wordsArray.listIterator(0);
        while (iterator.hasNext()) {
            String wordsArrayElement = iterator.next().toString();
            if (getASCIISum(textArrayElement) == getASCIISum(wordsArrayElement)) {
                //compare ASCII values with words.txt and textPane (notepad screen)
                s = textArrayElement;
                sameASCIIarr.add(wordsArrayElement);
            }
            //wordsArrayElement = iterator.next().toString();//adds words that have same ASCII values.
        }
        getTrueWord(s, sameASCIIarr, textPane);
    }
}
```

iterator ile gezildi.

Tasarım deseni ile birlikte iterator kullanarak listeyi gezdik. Ayrıca Single Transposition'a sahip olup olmadığı araştırılan currentWordChar dizisi de iterator ile gezildi. char olduğu için ayrıca StringCharacterIterator kullanılarak iterator tasarım deseni gerçekleştirildi.

```
public void getTrueWord(String currentWord, ArrayList arr, JTextPane textPane) throws BadLocationException {
    String tempWord = currentWord; //we keep currentWord
    char[] currentWordChar = currentWord.toCharArray();
    characterIterator = new StringCharacterIterator(currentWordChar.toString());
    iteratorWordArray = arr.iterator();

    String listIteratorElement;
    while (iteratorWordArray.hasNext()) {
        listIteratorElement = iteratorWordArray.next().toString();

        for (char c = characterIterator.first(); c != CharacterIterator.DONE; c = characterIterator.next()) { //loops up to sameASCII arr size

            for (int j = 0; j < currentWord.length() - characterIterator.getIndex() - 1; j++) {
                if (listIteratorElement.equals(currentWord)) //If the word in textpane and word in words.txt is the same
                {
                    findCh.changeGivenText(tempWord, listIteratorElement, textPane);
                }
                currentWord = tempWord;
                currentWordChar = currentWord.toCharArray();
```

Kullanıcı Arayüzünde Uygulaması:



3. Abstract Factory Tasarım Deseni

Birden fazla nesne ile çalışmak durumunda kaldığımızda, nesne ile istemci tarafını soyutlamak için kullanılır. Nesnelerin oluşumunu istemci tarafından ayırarak, karar verme koşulu olmadan, esnek ve geliştirilebilir bir yapı kurmamızı sağlar. Factory design pattern'de tek bir nesneye ait tek bir arayüz mevcutken, abstract factory'de farklı nesneler için farklı arayüzler mevcuttur.

Biz projede bu tasarım desenini Single Transposition, kelime arama ve bu kelimeyi istenilen başka bir kelime ile değiştirme durumları için kullandık.

→ Öncelikle abstract bir sınıf olan AbstractFactory class'ını oluşturduk. Burada Operation tipinde nesneler döndüren ve nesne tipini parametre olarak alan makeOperationObject() metodu bulunmakta. Bu metodun

implementasyonunda oluşturulacak nesnenin SingleTransposition mı yoksa FindAndChangeWord nesnesi mi olduğuna karar verilecek.

```
public abstract class AbstractFactory {

    abstract Operation makeOperationObject(String objectType);

}
```

→ Daha sonra Operation tipine bir interface oluşturduk. Burada makeOperation() metodu yer almaktadır. Parametre olarak textPane yani notepad ekranımızı alır. Buradaki amaç oluşturduğumuz farklı operasyonları oluşturduğumuz nesnelere göre implemente etmek.

```
public interface Operation {

    void makeOperation(JTextPane jTextPane);

}
```

→ Buna ek olarak AbstractFactory sınıfından extend eden bir OperationFactory sınıfı oluşturduk. Bu sınıf makeOperationObject()

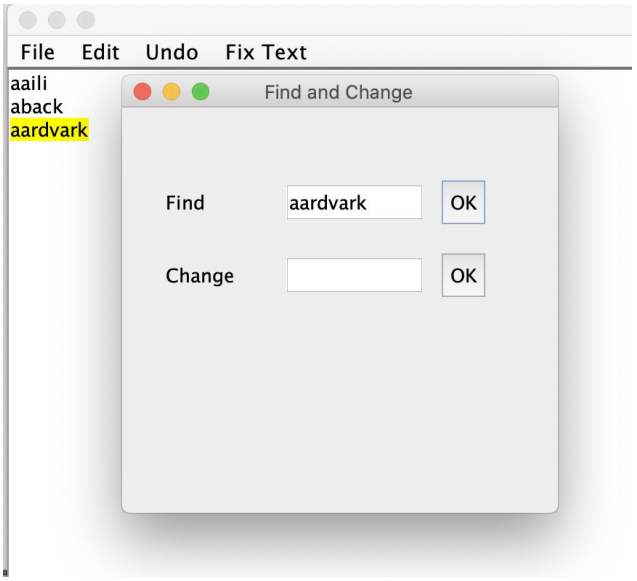
metodunu override ediyor. Aldığı object tipine göre nesnelerimizi oluşturup Operation tipinde döndürüyor.

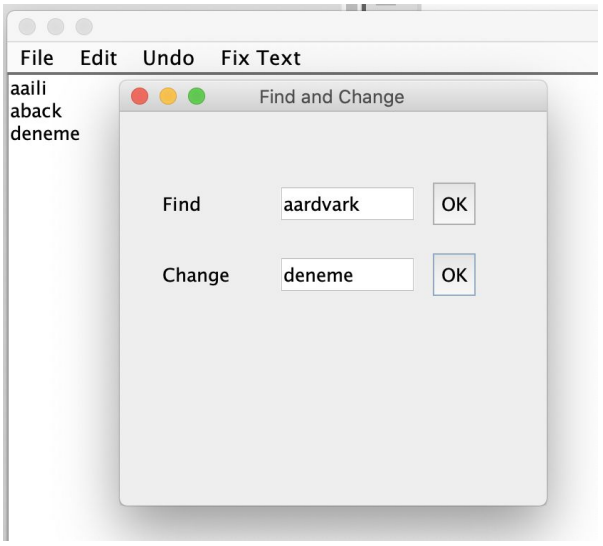
```
public class OperationFactory extends AbstractFactory {

    public Operation makeOperationObject(String objectType) {
        if (objectType.equalsIgnoreCase("Single Transposition")) {
            return (Operation) new SingleTransposition();
        } else if (objectType.equalsIgnoreCase("Find and Change Word")) {
            return (Operation) new FindAndChangeUI();

        }
        return null;
    }
}
```

→ Son olarak main metodumuzun yer aldığı Notepad.java sınıfında ise Single Transposition ve Find and Change Word için öncelikle Operation Factory tipinde operationFactory nesnesi tanımladık. Bu nesne üzerinden makeOperationObject() metodu ile istenilen nesneler yaratılıyor ve bu oluşturulan nesneler üzerinden makeOperation() metodları çağırılıyor. Böylece tasarım desenimizi uygulamış oluyoruz.

```
//SingleTransposition Button Action
Action SingleTransposition = new AbstractAction("Single Transposition") {
    @Override
    public void actionPerformed(ActionEvent e) {
        Operation singleTransposition = operationFactory.makeOperationObject("Single Transposition");
        singleTransposition.makeOperation(textPane);
    }
};

//FindAndChange Action. Instantiate new frame.
Action FindAndChange = new AbstractAction("Find and Change Word") {
    @Override
    public void actionPerformed(ActionEvent e) {
        Operation findAndChangeWord = operationFactory.makeOperationObject("Find and Change Word");
        findAndChangeWord.makeOperation(textPane);
    }
};
```

Kullanıcı Arayüzünde Uygulaması:





4. Strategy Tasarım Deseni

Strategy deseni, bir nesnenin herhangi bir operasyonu gerçekleştirmek için kullanabileceği farklı algoritmaları içeren farklı tipleri kendi içerisinde ele alarak kullanması yerine, kullanmak istediği fonksiyonelliği nasıl uygulandığını bilmesine gerek kalmaksızın sadece seçerek çalışma zamanında yürütmesine olanak tanımaktadır.

Bir durum gerçekleşeceği zaman birden fazla seçeneğimiz varsa istenilen veya gerçekleşmesi beklenen seçime(stratejiye) kolaylıkla ulaşmamızı sağlayan bir strateji sınıfına ihtiyaç duyulur. Seçeneklerimiz arttıkça kolaylıkla ekleme yapabilir veya durumlar arası geçişi kod fazlalığı olmadan gerçekleştirebiliriz.

Biz projede Strategy desenini dosya işlemleri için kullandık. Proje 1'de bu operasyonları tek bir sınıfta yapıyorduk. Fakat bu işlevsellik açısından tek bir sınıf üstünde fazla yük olmasına sebep oluyordu. Bu sınıfımızı ayırdık ve open file, create file, exit file, save file için farklı sınıflar oluşturduk.

→ Öncelikle MenuStrategy isminde bir interface oluşturduk. Bu interface'de executeProcess isminde bir metodumuz yer almakta. Bu metod bütün dosya işlemleri için ortak implemente edilecektir.

```
public interface MenuStrategy {

    public void executeProcess(JFileChooser fc, JTextPane jTextPane, File file);

}
```

→ Daha sonra StrategyContext isimli sınıfta ise MenuStrategy tipinde altığımız strategy'yi kullanacağız. Burada selectMenuItem() metodu yer almakta. Bu metod MenuStrategy sınıfındaki executeProcess metodunu kullanmaktadır.

```
public class StrategyContext {

    private MenuStrategy strategy;
    public void setMenuStrategy(MenuStrategy strategy){
        this.strategy = strategy;
    }
    public void selectMenuItem(JFileChooser fc, JTextPane jTextPane, File file){
        strategy.executeProcess(fc, jTextPane,file);
    }


}
```

→ Dosya işlemlerinden yeni dosya oluşturmak için CreateFileStrategy sınıfını oluşturduk ve birinci projede yaptığımız işlemleri tekrarladık. Bu sınıf MenuStrategy sınıfını implemente ederek executeProcess() methodunu override ediyor. Method içinde fileChooser ile dosya tipi

```
public class CreateFileStrategy implements MenuStrategy {

    @Override
    public void executeProcess(JFileChooser fc, JTextPane jTextPane, File file) {
        if (fc.showSaveDialog(null) == JFileChooser.APPROVE_OPTION) {
            FileWriter fw = null;
            try {
                fw = new FileWriter(fc.getSelectedFile().getAbsolutePath() + ".txt");
                jTextPane.write(fw);
                fw.close();

            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

seçilerek textPane'de (notepad ekranı) yazılanlar bu dosyaya write komutu ile yazılıyor.

→ Dosyadan çıkmak için ExitFileStrategy sınıfını oluşturduk. Bu sınıf da MenuStrategy sınıfını implemente ediyor ve executeProcess() metodunu override ediyor. Kendi işlevine göre de implemente ettik.

```
public class ExitFileStrategy implements MenuStrategy {

    @Override
    public void executeProcess(JFileChooser fc, JTextPane jTextPane, File file) {
        System.exit(0);
    }

}
```

→ Notepad'de var olan dosyayı açmak için OpenFileStrategy sınıfını oluşturduk. Bu sınıf da MenuStrategy sınıfını implemente ediyor ve executeProcess() metodunu override ediyor.

```
public class OpenFileStrategy implements MenuStrategy {

    @Override
    public void executeProcess(JFileChooser fc, JTextPane jTextPane, File file) {
        String text = "";
        JFileChooser j = new JFileChooser();

        int r = j.showOpenDialog(null);

        if (r == JFileChooser.APPROVE_OPTION) {

            File fi = new File(j.getSelectedFile().getAbsolutePath());
            file = fi;
            try {

                FileReader fr = new FileReader(fi);
                int i = fr.read();

                while (i != -1) {
                    text += (char) i;
                    i = fr.read();
                }
                System.out.print(text);

            } catch (Exception evt) {

            }
        }
        jTextPane.setText(text);
        Notepad.file = file;
    }

}
```

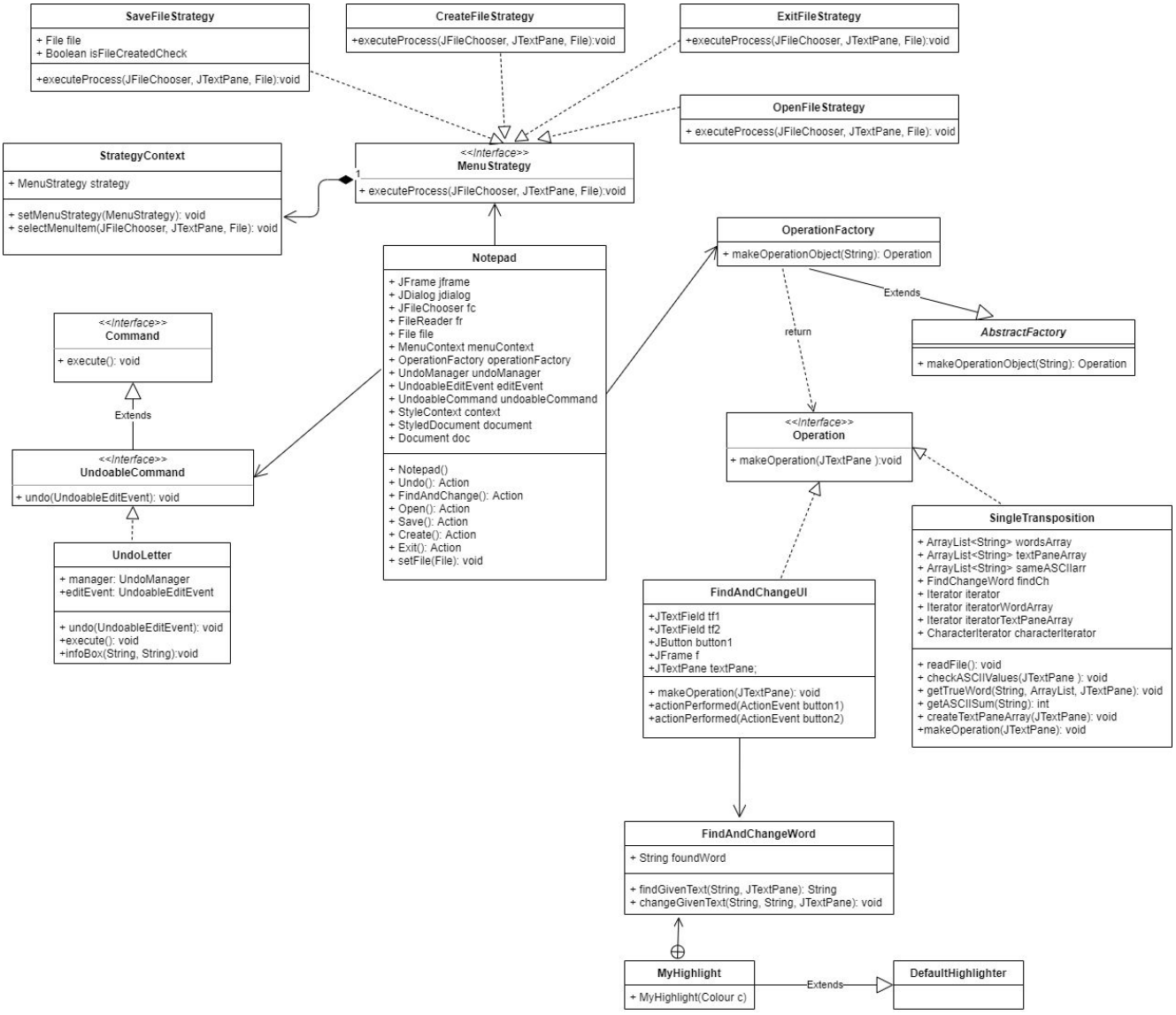
→ Dosyayı kaydetmek için SaveFileStrategy sınıfını oluşturduk. Bu sınıf da MenuStrategy sınıfını implemente ediyor ve executeProcess() metodunu override ediyor.

```
public class SaveFileStrategy implements MenuStrategy {


    @Override
    public void executeProcess(JFileChooser fc, JTextPane jTextPane, File file) {
        try {
            FileWriter fw = null;


            fw = new FileWriter(file.getAbsolutePath());

            fw.write(jTextPane.getText()); //writes everything in the file to textPane
            fw.close();
        } catch (IOException ex) {
            Logger.getLogger(SaveFileStrategy.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

→ mainin bulunduğu Notepad sınıfında ise MenuContext tipinde oluşturduğumuz menuContext nesnesi üzerinden ilk olarak setMenuStrategy() metod ile hangi sınıf çağırılmak isteniyorsa onu oluşturuyoruz. Seçimi yaptıktan sonra selectMenuItem() metodu üzerinden file chooser, textpane ve file objelerimizi bu metoda yollayarak gerekli işlemleri yapmasını sağlıyoruz. Burada Open, Create ve Exit işlemleri için file'ı göndermemize gerek yok. Bu sebeple null olarak parametre geçtik.

```
//Open File Action
Action Open = new AbstractAction("Open") {
    @Override
    public void actionPerformed(ActionEvent e) {
        //menu context ile gerekli strateji nesnesini yaratma.
        if (fc.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
            menuContext.setMenuStrategy(new OpenFileStrategy());
            menuContext.selectMenuItem(fc, textPane, null);
        }
    }
};

//Save File Action
Action Save = new AbstractAction("Save") {
    @Override
    public void actionPerformed(ActionEvent e) {
        //menu context ile gerekli strateji nesnesini yaratma.
        menuContext.setMenuStrategy(new SaveFileStrategy());
        menuContext.selectMenuItem(fc, textPane, file);
    }
};

//Create File Action
Action Create = new AbstractAction("Create") {
    @Override
    public void actionPerformed(ActionEvent e) {
        //menu context ile gerekli strateji nesnesini yaratma.
        menuContext.setMenuStrategy(new CreateFileStrategy());
        menuContext.selectMenuItem(fc, textPane, null);
    }
};

//Exit File Action
Action Exit = new AbstractAction("Exit") {
    @Override
    public void actionPerformed(ActionEvent e) {
        //menu context ile gerekli strateji nesnesini yaratma.
        menuContext.setMenuStrategy(new ExitFileStrategy());
        menuContext.selectMenuItem(fc, textPane, null);
    }
};
```

5. UML


SaveFileStrategy
+ File file
+ Boolean isFileCreatedCheck
+executeProcess(JFileChooser, JTextPane, File):void
CreateFileStrategy
+executeProcess(JFileChooser, JTextPane, File):void
ExitFileStrategy
+executeProcess(JFileChooser, JTextPane, File):void
OpenFileStrategy
+ executeProcess(JFileChooser, JTextPane, File): void
StrategyContext
+ MenuStrategy strategy
+ setMenuStrategy(MenuStrategy): void
+ selectMenuItem(JFileChooser, JTextPane, File): void
<<Interface>>
MenuStrategy
+ executeProcess(JFileChooser, JTextPane, File):void
1
Notepad
+ JFrame jframe
+ JDialog jdialog
+ JFileChooser fc
+ FileReader fr
+ File file
+ MenuContext menuContext
+ OperationFactory operationFactory
+ UndoManager undoManager
+ UndoableEditEvent editEvent
+ UndoableCommand undoableCommand
+ StyleContext context
+ StyledDocument document
+ Document doc
+ Notepad()
+ Undo(): Action
+ FindAndChange(): Action
+ Open(): Action
+ Save(): Action
+ Create(): Action
+ Exit(): Action
+ setFile(File): void
OperationFactory
+ makeOperationObject(String): Operation
Extends
AbstractFactory
+ makeOperationObject(String): Operation
return
<<Interface>>
Command
+ execute(): void
Extends
<<Interface>>
UndoableCommand
+ undo(UndoableEditEvent): void
UndoLetter
+ manager: UndoManager
+editEvent: UndoableEditEvent
+ undo(UndoableEditEvent): void
+execute(): void
+infoBox(String, String):void
<<Interface>>
Operation
+ makeOperation(JTextPane):void
SingleTransposition
+ ArrayList<String> wordsArray
+ ArrayList<String> textPaneArray
+ ArrayList<String> sameASCIIarr
+ FindChangeWord findCh
+ Iterator iterator
+ Iterator iteratorWordArray
+ Iterator iteratorTextPaneArray
+ CharacterIterator characterIterator
+ readFile(): void
+ checkASCIIValues(JTextPane): void
+ getTrueWord(String, ArrayList, JTextPane): void
+ getASCIISum(String): int
+ createTextPaneArray(JTextPane): void
+makeOperation(JTextPane): void
FindAndChangeUI
+JTextField tf1
+JTextField tf2
+JButton button1
+JFrame f
+JTextPane textPane;
+ makeOperation(JTextPane): void
+actionPerformed(ActionEvent button1)
+actionPerformed(ActionEvent button2)
FindAndChangeWord
+ String foundWord
+ findGivenText(String, JTextPane): String
+ changeGivenText(String, String, JTextPane): void
MyHighlight
+ MyHighlight(Colour c)
Extends
DefaultHighlighter