מרכז ההדרכה 2000
תמיכה ועדכונים
עדכון מס' 67
אוגוסט 2002

מימוש מודל פקודות במערכות עצמים: Command Pattern

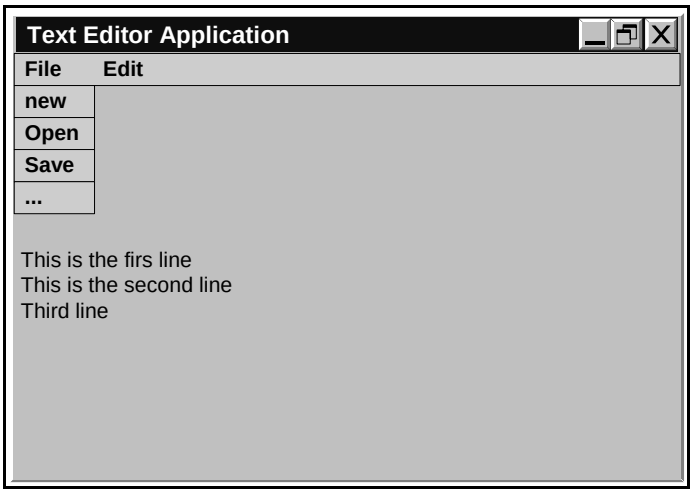
מבוא

מודל פקודות משתמש, בדומה למודל אירועים, עוסק בצד הממשק למשתמש (לרוב ממשק גרפי, אך לא בהכרח) : במודל זה, המשתמש מעביר פקודה לתכנית באמצעות מרכיב מסויים בממשק המשתמש - לדוגמא תפריט או תיבת דו-שיח. המטרה היא כמובן לספק מודל מודולרי ויעיל ככל שניתן כפתרון כללי לבעיה.

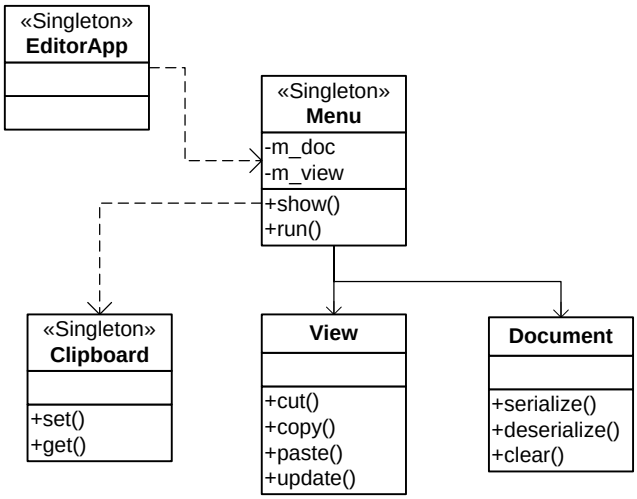
במאמר זה נסקור את ה- Command כ- Pattern למימוש מודל פקודות יעיל ומודולרי במערכת מונחית עצמים. תוכן נושא זה מבוסס על סעיף "Command Pattern" שבפרק 16 בספר "***C++ - מדריך מקצועי***", בהוצאת "**מרכז ההדרכה 2000**".

Command Pattern

<u>בעיה</u>
במסגרת בניית יישום עריכת טקסט (Editor), נדרש לתכנן מחלקת תפריט (**Menu**) שתציג פריטים במבנה תפריט למשתמש, ובתגובה לבחירת המשתמש בפריט כלשהו תופעל פונקצית התגובה המתאימה.



פתרון שגוי : המחלקה Menu (Singleton) תחזיק מצביעים לעצמים המתאימים ביישום :



- Document מייצגת את נתוני המסמך ביישום
- View מייצגת את חלון התצוגה
- Clipboard מייצגת את לוח ההעתקה (Singleton)
- EditorApp מייצגת את מחלקת היישום (Siingleton)

Menu כוללת את הפונקציה **run()** הבודקת איזה פריט נלחץ, ובהתאם מבצעת את הפעולה. לדוגמא, מבנה של יישום עריכת טקסט (Editor) ייראה כך :

```
class Document
{
        ...
public:
        void serialize(string file_name="") ;
        void deserialize(string file_name="") ;
        void clear() ;
};

class View
{
        ...
public:
        void update();
        string copy() ;
        string cut() ;
        void paste(string) ;

};

class Clipboard
{
        ...
public:
        static Clipboard& instance() { static Clipboard c; return c; }
        void set(string ) ;
        string get() ;
};


class Menu    // Singleton
{
        Document    *m_doc;
        View        *m_view;
        string      file_name;
```

```
        enum {NEW, OPEN, SAVE, SAVE_AS, EXIT, CUT, COPY, PASTE};
        Menu();
  public:
        static Menu& instance() ;   // Singleton
        int             show();         // show menu and return user selection
        void            run()           // main function in menu
        {
          while(true)  // main loop of application
          {
                int selection = show();
                switch(selection)
                {
                        case NEW:
                                m_doc->clear();
                                m_view->update();
                                break;
                        case OPEN:
                                m_doc->clear();
                                file_name = FileDialog::show(FileDialog::F_READ);
                                m_doc->deserialize(file_name);
                                m_view->update();
                                break;
                        case SAVE:
                                m_doc->serialize(file_name);
                                break;
                        case SAVE_AS:
                                file_name = FileDialog::show(FileDialog::F_WRITE);
                                m_doc->serialize(file_name);
                                break;
                        case CUT:
                                Clipboard::instance().set( m_view->cut() );
                                m_view->update();
                                break;
                        case COPY:
                                Clipboard::instance().set( m_view->copy() );
                                break;
                        case PASTE:
                                m_view->paste ( Clipboard::instance().get() );
                                m_view->update();
                                break;
                        case EXIT:
                                return;
                        ...
                }
          }
        }
};
```

לפתרון זה מספר חסרונות בסיסיים:

- **יעילות**: הפתרון אינו יעיל - בכל בחירת משתמש מבוצע חיפוש של הכניסה המתאימה במשפט ה-switch בסיבוכיות O(n), כאשר n הוא מספר האפשרויות בתפריט. למשל, אם יש 100 פריטים בתפריט, בחירת משתמש תגרום לסריקה של 50 אפשרויות בממוצע.
- **מודולריות**: הפתרון גם אינו מודולרי - קוד היישום העיקרי נמצא באופן ריכוזי במחלקת התפריט, מה שהופך אותה לבלתי ניתנת לשימוש חוזר ביישום אחר. ובכלל, המחלקה Menu משמשת גם כמנהלת התכנית, מלבד תפקיד הצגת התפריט.
- **דינמיות**: הפתרון אינו מאפשר לקבוע בזמן ריצה את התגובה לבחירה מסוימת - הוא אינו מאפשר להוסיף פריטים חדשים, למחוק פריטים ישנים ולשנות את פונקציות התגובה.

יש לשים לב שעקב חוסר המודולריות של הפתרון, יש קושי בתחזוקה של קוד ה- Menu. כמו כן יש קושי

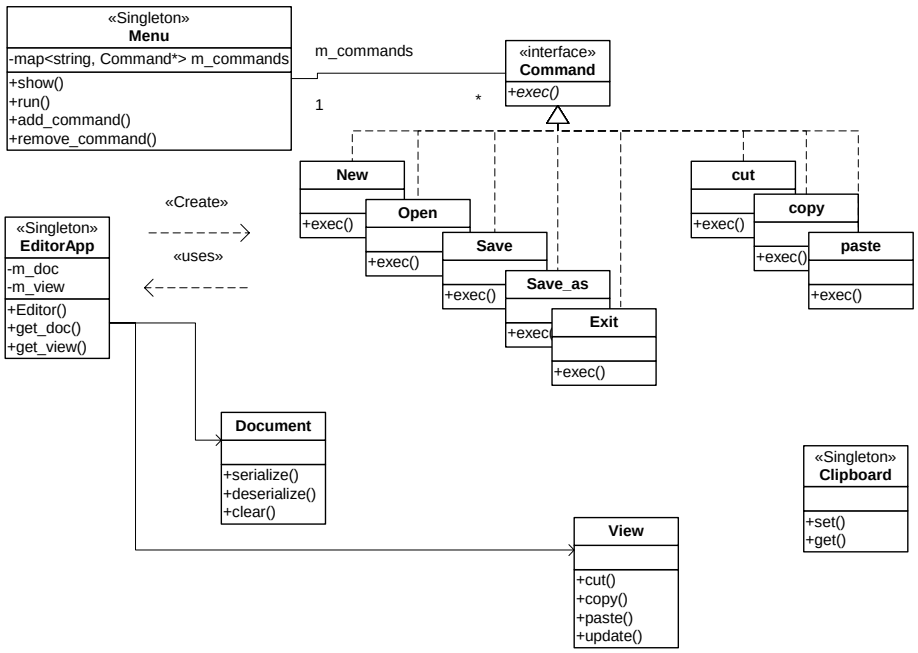
בהוספת מנגנונים נוספים כגון: Undo / Redo, טיפול במספר מסמכים בו-זמנית (**MDI**, Multiple Document Interface).

<u>פתרון</u>

שימוש ב- **Command Pattern** :

- ממשק בשם **Command** הכולל פונקציה אבסטרקטית יחידה - **exec**()
- מחלקת **Menu** מחזיקה טבלת מצביעים ל- Command עפ"י מחרוזות (map<stirng, Command *>). מחלקה זו כללית ואינה מודעת כלל לסוג היישום שאנו מפתחים.
- כל פקודה בתפריט מיוצגת ע"י מחלקה הממשת את **Command**
- הפקודות המסוימות מחזיקות מצביעים לעצמים המתאימים: View, Document
- כאשר המשתמש בוחר בפריט בתפריט, **Menu** מוצאת את הפקודה המתאימה עפ"י המחרוזת, ומפעילה את הפונקציה הוירטואלית שלה **exec**().

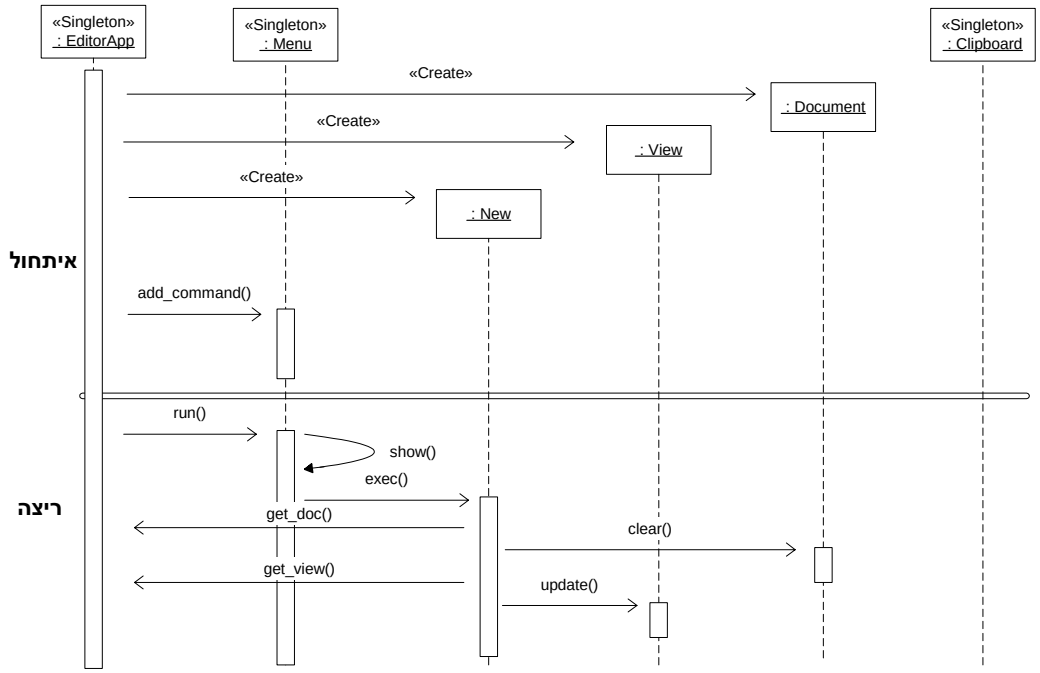
תרשים המחלקות:



ליישום שני שלבים: בשלב האתחול, מחלקת היישום, EditorApp, יוצרת עצמי Command מתאימים ומכניסה אותם לעצם התפריט. בסיום האתחול היא קוראת לפונקציה run() של התפריט.

השלב השני הוא שלב הריצה, בו המשתמש בוחר בפריטים מהתפריט והפקודה המתאימה מבוצעת: הפונקציה select() מחזירה כעת את מחרוזת הפריט, מוצאים בטבלה את הפקודה המתאימה וקוראים לפונקציה הוירטואלית exec() שלה.

תרשים ה- Sequence הבא מתאר את שלב האתחול ושלב הריצה עבור הפקודה File / New :



קוד הממשק **Command** :

```
class Command
{
public:
        virtual void exec() = 0;
        virtual ~Command() {}
};
```

ה- destructor מוגדר כוירטואלי בכדי ששחרור העצמים הפולימורפי מתוך Menu יבוצע באופן תקין.

קוד המחלקה **Menu** :

```
class Menu
{
        map<string, Command*>  m_commands;
        Menu();
        ~Menu();
public:
        static Menu& instance() ;  // Singleton
        void            add_command(string submenu, string item, Command* c);
        void            remove_command(string item);
        string          show();
        void            run()
        {
                while(true) // main loop of application
                {
                        string selection = show();
                        m_commands[selection]->exec();
                }
        }
};
```

קוד המחלקה **New** :

```
class New : public Command
{
public:
        virtual void exec();
};

void New::exec()
{
        EditorApp::instance().get_doc()->clear();
        EditorApp::instance().get_view()->update();
}
```

קוד המחלקה **EditorApp** :

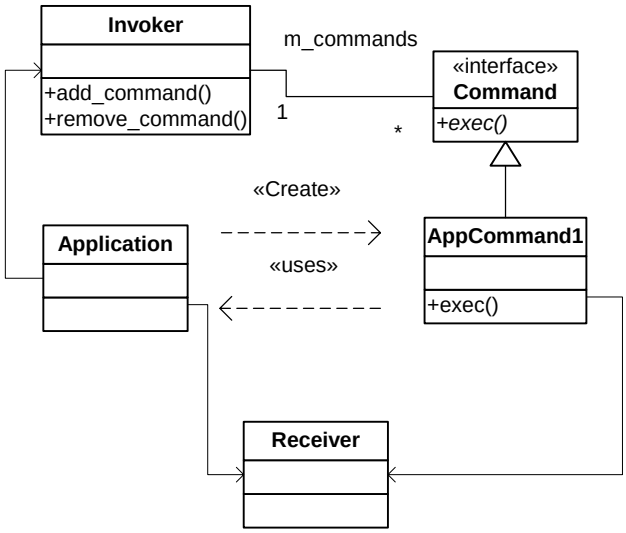
```
class EditorApp
{
public:
        Document*   get_doc();              // access for the Command objects
        View*       get_view();             // access for the Command objects
        static EditorApp& instance();       // Singleton
private:
        Document    *m_doc;
        View        *m_view;
        EditorApp()
        {
                m_doc = new Document;
                m_view = new View;

                Menu::instance().add_command( "File", "New", new New);
                Menu::instance().add_command( "File", "Save", new Save);
                Menu::instance().add_command( "File", "Save as", new Save_as);
                ...
        }
};
```

כתרגיל הבנה, נסה/י לכתוב את קוד המחלקות **Save**, **Save_as**.

<u>הכללה</u>

Command הוא Pattern שימושי במצבים בהם נדרש להעביר פקודות משתמש - בד"כ מרכיב ממשק משתמש גרפי (כגון : תפריט, סרגל כלים) - לעצמים שונים ביישום :



תיאור המרכיבים :

Command - ממשק בסיס קבוע לכל הפקודות בכל היישומים

AppCommand1 - פקודה מסוימת של היישום

Application - מחלקת יישום הלקוח

Invoker - מחלקה המחזיקה את טבלת ה- Commands ומבצעת קריאה לפונקציה **exec()** של ה- Command הנבחר.

Receiver - מחלקת העצם שעליו מופעלת הפקודה.

יתרונות ה- **Command** :

- **יעילות** : מכיוון שהפריטים כעת מוחזקים בטבלה עפ"י המחרוזת, זמן החיפוש של פריט הוא כעת log(n). כלומר, אם בתפריט 100 פריטים כמו קודם, זמן החיפוש יהיה כעת בממוצע פחות מ- 7.
- **מודולריות** : מבנה המחלקות הוא כעת מאוד מודולרי - קיימת הפרדה בין המחלקה המפעילה את הפקודה (Menu) לבין הפקודה (Command) :
  - Menu אינו "מכיר" כלל את טיפוסי הפקודות הספציפיים, ולכן אינו תלוי בהם (הוא אף אינו צריך לעבור הידור מחודש בשינוי או הוספה של פקודה), מה שהופך אותו לכללי ולשימוש חוזר.
  - פקודה "יודעת" מה צריך לבצע, ולכן שינוי או הוספת פקודה גורמת לשינוי במחלקת הפקודה המסוימת ובקוד הלקוח (EditorApp) בלבד.
- **דינמיות** : מבנה ה- Command מאפשר הוספה והסרה של פקודות בזמן ריצה.

בנוסף, המבנה המודולרי של Command מאפשר הוספת מנגנונים כגון :

- **Undo / Redo** - ניתן בקלות יחסית לממש מנגנון זה ע"י החזקת תור היסטוריה של פקודות שבוצעו. במצב זה תוגדר בממשק Command הפונקציה unexec() שתבצע - במידה וניתן - ביטול של exec().
- **הרחבה ליישום מרובה מסמכים** (**MDI**) - בדוגמא שלעיל, EditorApp יכולה להחזיר מספר זוגות של **מסמך-תצוגה** (Document-View), כאשר זוג אחד הוא כרגע הפעיל. פונקציות הגישה get_doc() ו- get_view() יחזירו מצביעים לעצמים הנוכחיים.
- **הגדרת פקודות מאקרו** - אלו הן קבוצות של פקודות הניתנות לביצוע כפקודה אחת. ניתן בפשטות להגדיר פקודה המחזיקה רשימה של פקודות אחרות לביצוע. כאן ניתן לעשות שימוש ב- **Composite Pattern** לייצוג מחלקת פקודת-מאקרו כללית המכילה רשימת פקודות רגילות.

- **אפשרויות וריאציות** :
  - מחלקות ה- Command יכולות להחזיק מידע נוסף ולהגדיר פונקציות נוספות עפ"י הצורך.
  - ה- Invoker יכול להחזיק את ה- Commands גם עפ"י מפתח מספרי (Enum)
  - תחת מגבלות מסוימות, ה- Invoker יכול להחזיק את ה- Commands עפ"י מזהים מספריים רצופים, כך ששליפת הפקודה המתאימה מהטבלה מבוצעת ב- O(1).

- **שימושים ידועים** :

1. MS-MFC מחזיקה טבלת מיפוי (Message Map) של מזהים מספריים לפונקציות. טבלה זו היא קבועה בזמן ריצה, והיא נבנית ע"י מקרואים בזמן הידור. לדוגמא, אם הגדרנו מחלקת תצוגה, CMyView, כיורשת ממחלקת הספרייה CView, כך מוגדר הטיפול בהדפסה (ממומש ב- CView) ובבחירת צבע אדום מתפריט הצבעים :

```
BEGIN_MESSAGE_MAP(CMyView, CView)
    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
    ON_COMMAND(ID_COLOR_RED, OnColorRed)
END_MESSAGE_MAP()
```

הסבר : המאקרו BEGIN_MESSAGE_MAP פורש טבלה בזמן הידור, הקובעת איזו פונקציה תופעל בתגובה לאיזו בחירה של פריט. פונקצית תגובה זו יכולה להיות של מחלקת הבסיס או של המחלקה

הנגזרת.

כפי שניתן לראות, כאן נבחרה הגישה של מיפוי הפקודה היישר לפונקצית תגובה החברה במחלקה שבה מעונינים לטפל בהודעה.

2. Java עושה שימוש ב- **Observer** עבור פקודות משתמש במסגרת מודל האירועים (Event Model) שלה. בכדי לקבל אירועי פקודה, יש לממש את הממשק **ActionListener** ולממש את הפונקציה ()**actionPerformed** המוגדרת בו. אולם כאן אין אפשרות לדעת איזו פקודה נבחרה - יש לבדוק את כל האפשרויות:

```
public class Editor extends Frame
        implements ActionListener
{
        ...
        public void actionPerformed(ActionEvent event)
        {
                if (event.getActionCommand().equals("New"))
                                ...
                else if (event.getActionCommand().equals("Open"))
                                ...
                else if (event.getActionCommand().equals("Save"))
                                ...
                else if (event.getActionCommand().equals("Exit"))
                                ...
                else if (event.getActionCommand().equals("Cut"))
                                ...
                else if (event.getActionCommand().equals("Copy"))
                                ...
        }
}
```

כמובן, המתכנת יכול להשלים את החסר ב- Java ע"י מימוש עצמאי של ה- Command: ניתן להגדיר מחלקה המחזיקה טבלה הממפה מחרוזות ל- Command, לאתחל אותה באתחול התפריט, ומתוך הפונקציה ()actionPerformed להפעיל את ה- Command המתאים:

```
interface Command
{
        public void exec();
}
public class Editor extends Frame
        implements ActionListener
{
        Hashtable       m_commands = new Hashtable(100);  // command table
        ...
        public void actionPerformed(ActionEvent event)
        {
                String item  = event.getActionCommand();
                (Command) m_commands.get(item).exec();
        }
}
```

3. **עצמי פונקציות** ב- C++ (**Function Objects**) - כלומר עצמים ממחלקות המבצעות Overloading לאופרטור הקריאה לפונקציה ()operator - הן למעשה וריאציה של Command: אלגוריתמים שונים מקבלים אותן כפרמטרים (ובעיקר עצמי פונקציה בוליאניים - פרדיקטים), והם משמשים כחלופה מונחית-עצמים טובה יותר למצביעים לפונקציות.

לדוגמא, אם נתונה מחרוזת

```
string word;
```

ניתן להגדיר מחלקה המייצגת עצם פונקציה:

```
struct Lower
```

```
{
        void operator()(char &c) { c=tolower(c); }
};
```

ולהשתמש בה כך להפיכת המילה word ל- lower case :

```
for_each(word.begin(), word.end(), Lower());
```

כל הזכויות שמורות © מרכז ההדרכה 2000