

# 5. Шаблон Команда (Command)

---

ЛЕКЦИОНЕН КУРС: ШАБЛОНИ ЗА ПРОЕКТИРАНЕ

ГЛ. АС. Д-Р ЕМИЛ ДОЙЧЕВ

# Общи сведения

---

- ✓ **Вид:** Поведенчески за обекти
- ✓ **Цел:** Капсулира заявките във вид на обекти, така че клиентите да могат да се параметризират с различни заявки, да подреждат в опашки или да документират заявки, или да поддържат операции по отмяна на действие или възстановяването му.
- ✓ **Известен и като:** Действие (Action), Транзакция (Transaction)

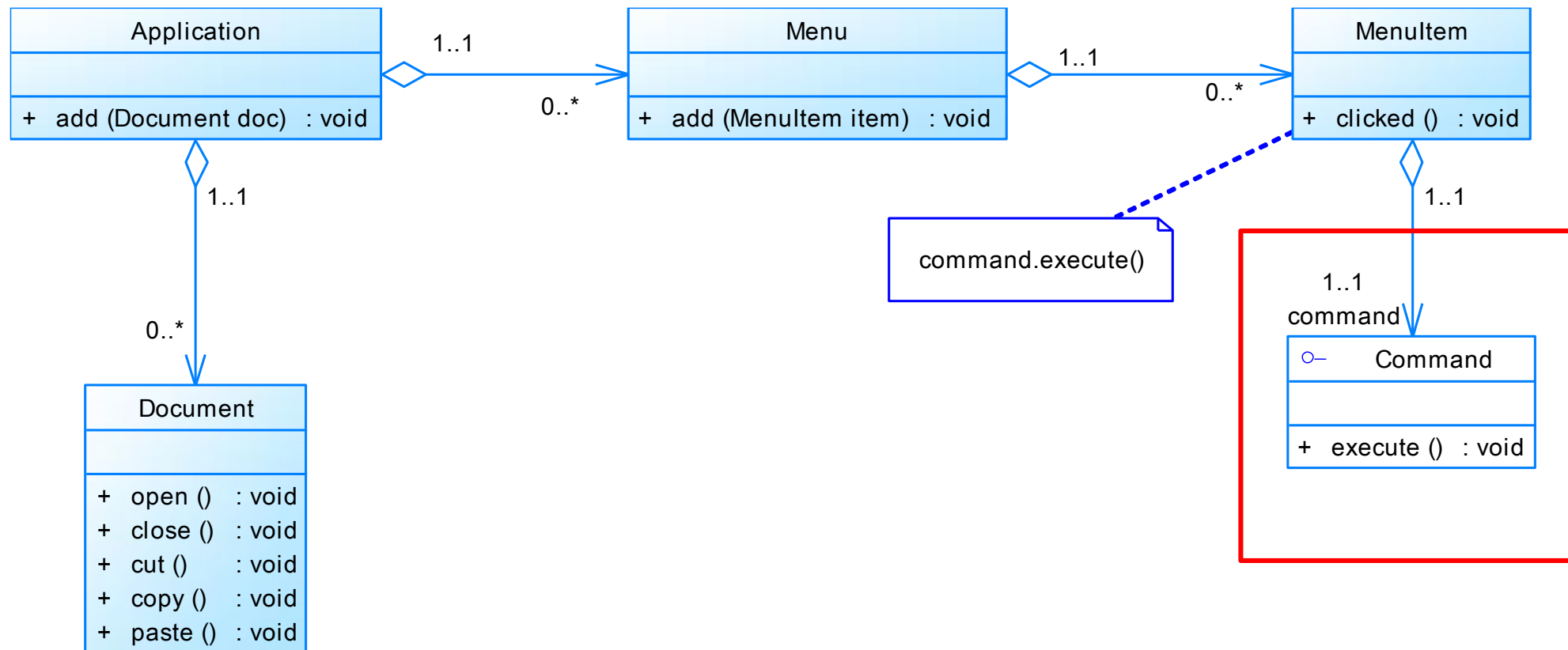
# Мотивация

---

- ✓ Понякога е необходимо да се изпращат заявки към някакви обекти без да се знае нищо за изискваната операция или получателя на заявката.
- ✓ **Пример:** набор от инструменти за графичен потребителски интерфейс (GUI) включват обекти, като бутони и менюта, изпълняващи заявки в отговор на действия на потребителя.
- ✓ Тези обекти не могат да имплементират заявката изрично в себе си – само приложенията, които използват набора инструменти, знаят какво трябва да се направи в отговор на заявката.
- ✓ Шаблонът *Команда* дава възможност на обектите от набора инструменти да правят заявки към неопределени обекти в приложението, като превръщат самата заявка в обект.

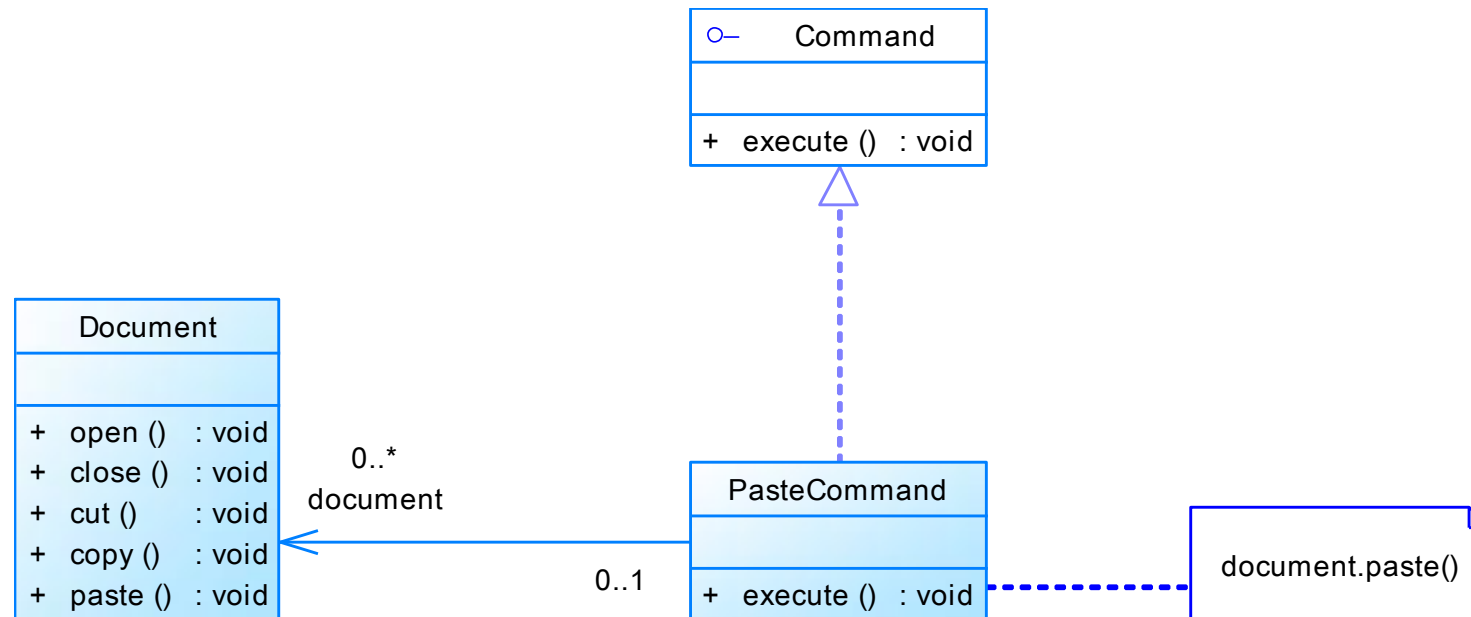
# Мотивация

- ✓ Пример за използване на шаблона Команда в меню



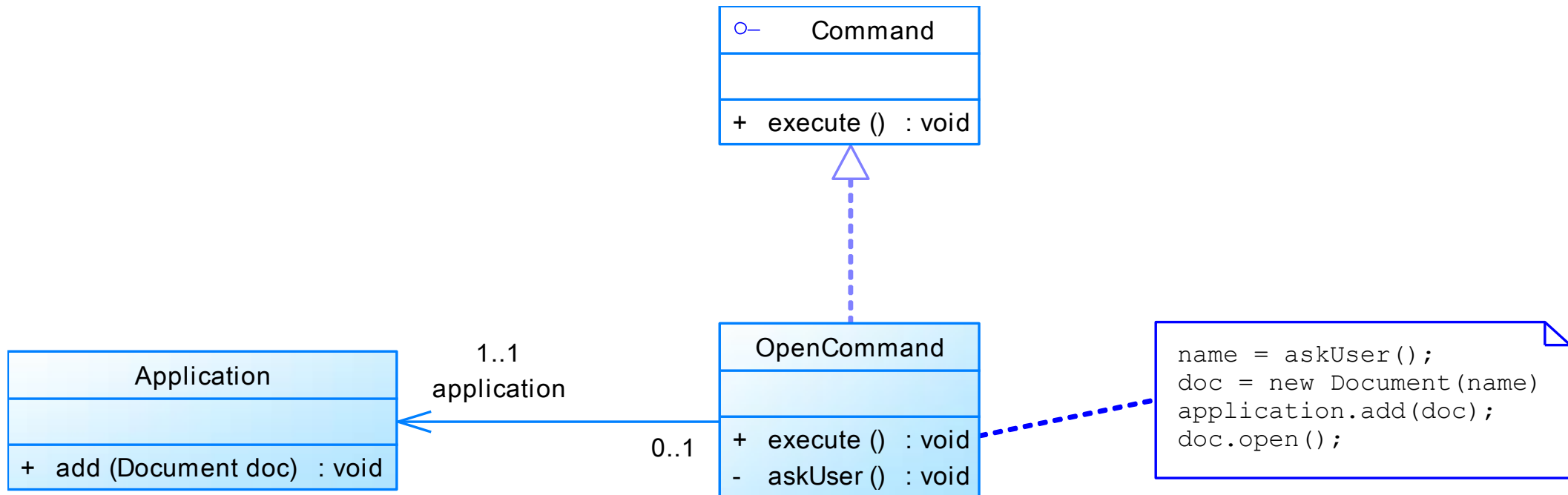
# Мотивация

- ✓ Подкласовете на *Command* съхраняват получателя на заявката (в примера – *Document*) и извършват една или повече операции с него.



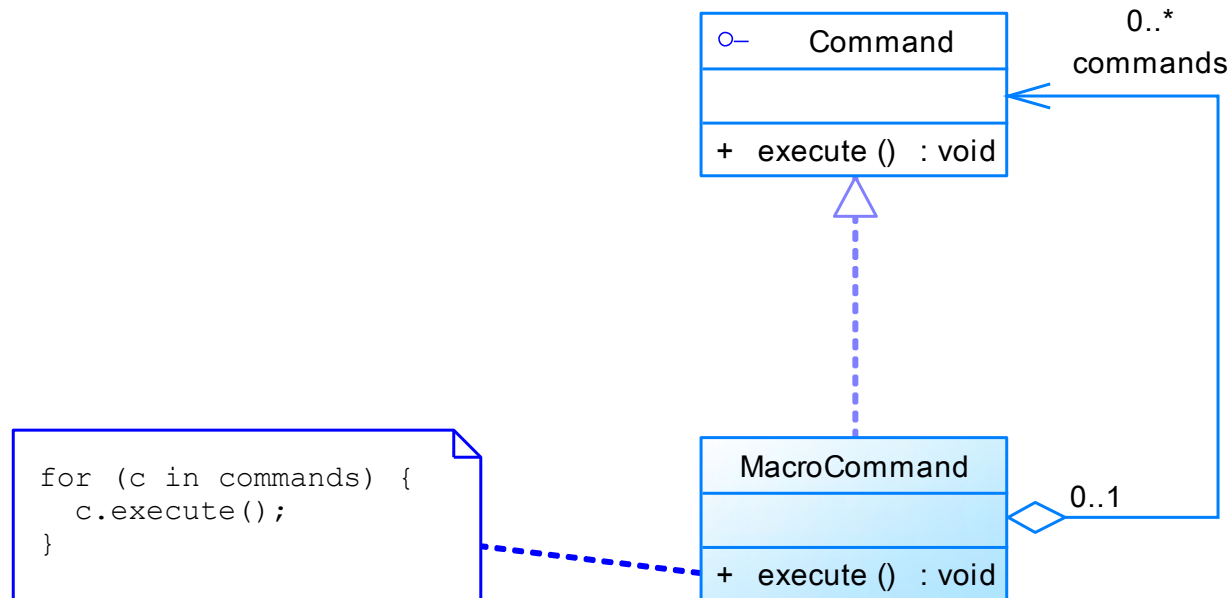
# Мотивация

- ✓ Пример за командата *Open*, която изисква от потребителя име на документа и след това създава документ с това име



# Мотивация

- ✓ Понякога един MenuItem обект трябва да изпълни поредица от команди – например центрирането на страница в нормален размер може да се извърши с поредица от две команди: *CenterDocumentCommand* и *NormalSizeCommand*. За целта може да дефинираме клас *MacroCommand*, с който да дадем възможност за изпълнение на неограничен брой команди.



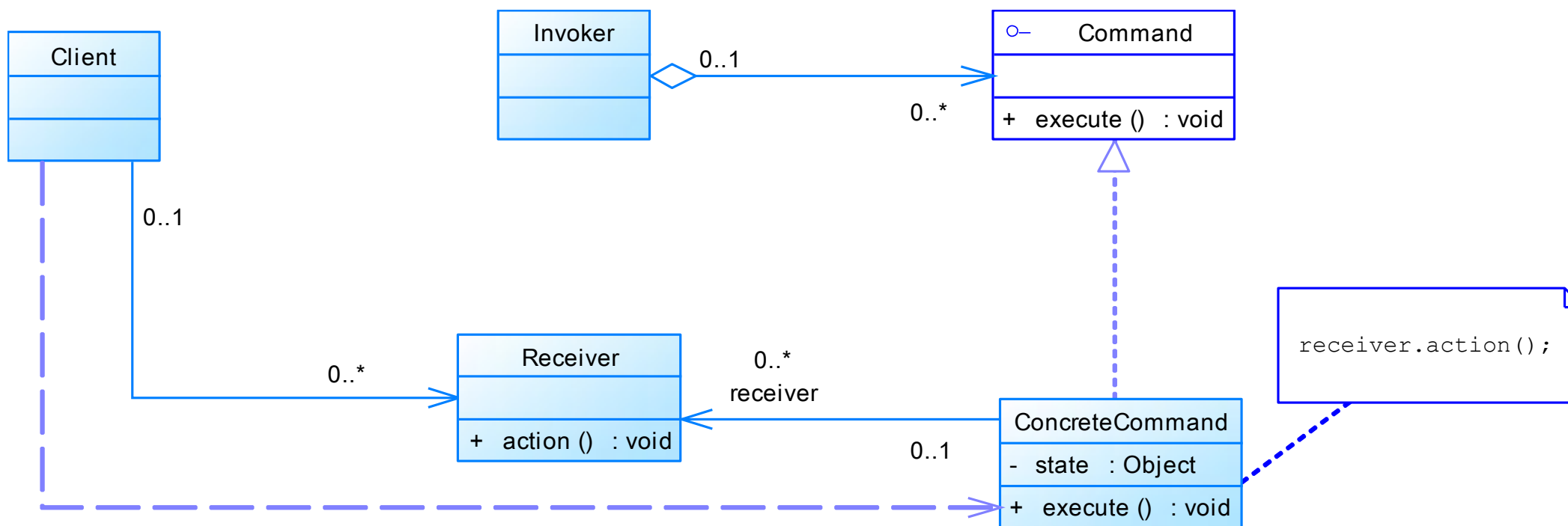
# Приложимост

---

- ✓ Шаблонът *Команда* се използва в следните случаи:
  - Реализация на функция за обратно извикване (callback) – т.е. функция, регистрирана някъде, за да се извика на по-късен етап. Командите са ОО заместители на обратното извикване.
  - Задаване, поддръждане в опашка и изпълняване на заявки по различно време.
  - Поддържане на отмяна на действия – операцията *execute* на командата може да съхранява състояния, за да има възможност за отмяна на действието на командата. В този случай към интерфейса на *Command* се добавя операция *unexecute*, която да отменя действието на предишното извикване на *execute*.
  - Поддържане на дневник на промените, така че да могат да се приложат отново при срив на системата. Интерфейса *Command* трябва да се разшири с операции по зареждане и съхраняване.



# Структура



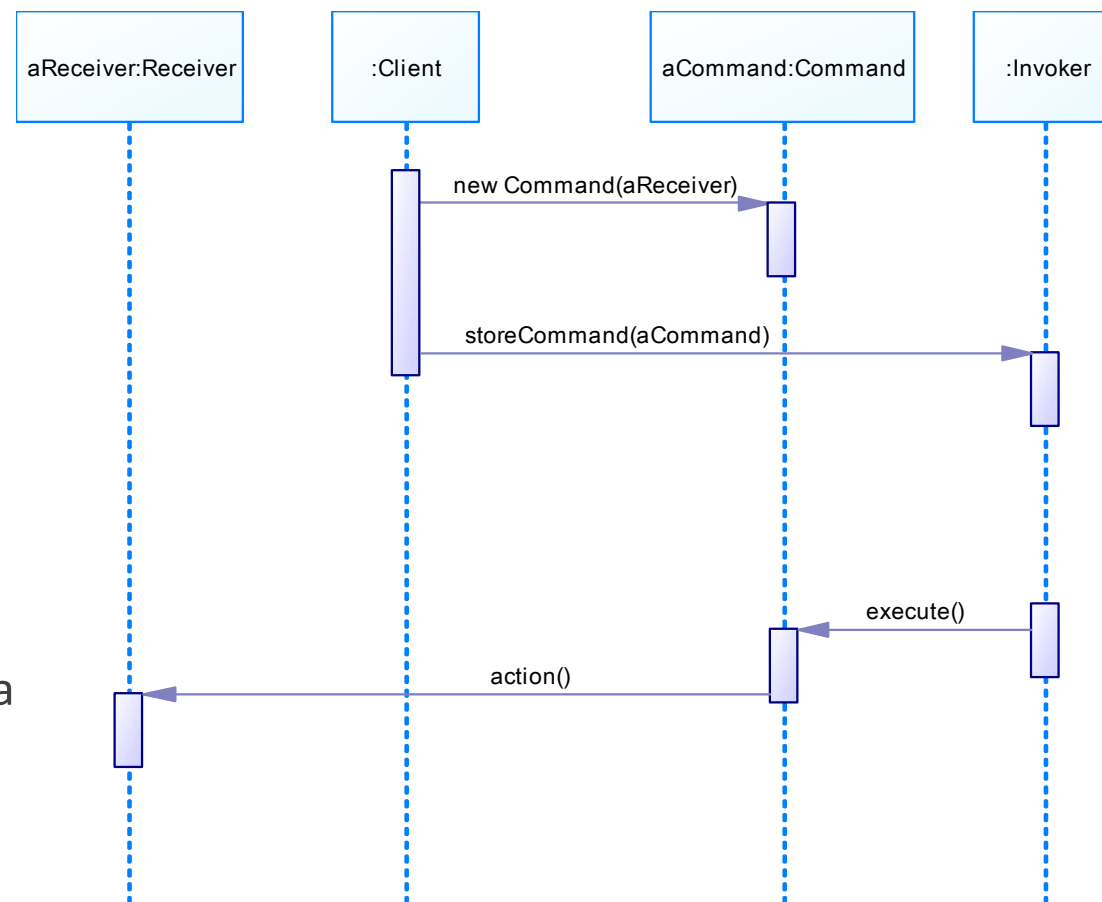
# Участници

---

- ✓ **Command** – декларира интерфейс за изпълнение на операция.
- ✓ **ConcreteCommand** (PasteCommand)
  - дефинира връзка между *Receiver* обект и действие
  - имплементира *execute* чрез извикване на съответните операции на *Receiver*
- ✓ **Client** (Application) – създава *ConcreteCommand* обект и задава получателя му.
- ✓ **Invoker** (MenuItem) – указва на командата да изпълни заявката.
- ✓ **Receiver** (Document, Application) – знае как да извърши операциите, асоциирани с изпълнението на заявката. Всеки клас може да служи за получател.

# Взаимодействия

1. Клиентът създава *ConcreteCommand* и задава получателя му.
2. *Invoker* обектът съхранява *ConcreteCommand* обекта.
3. *Invoker* обектът издава заявка чрез извикване на *execute* на командата.
4. *ConcreteCommand* обектът извиква операции на своя получател за да изпълни заявката.



# Следствия

---

- ✓ Класът *Command* разделя обекта, извикващ операцията, от този, който знае как да я извърши.
- ✓ Командите могат да бъдат манипулирани и разширявани както всеки друг обект.
- ✓ Може лесно да се сглобява множество от команди в по-големи сложни команди (макро команди)
- ✓ Добавянето на нови команди е лесно, защото не се налага да се променят съществуващи класове.

# Имплементация

---

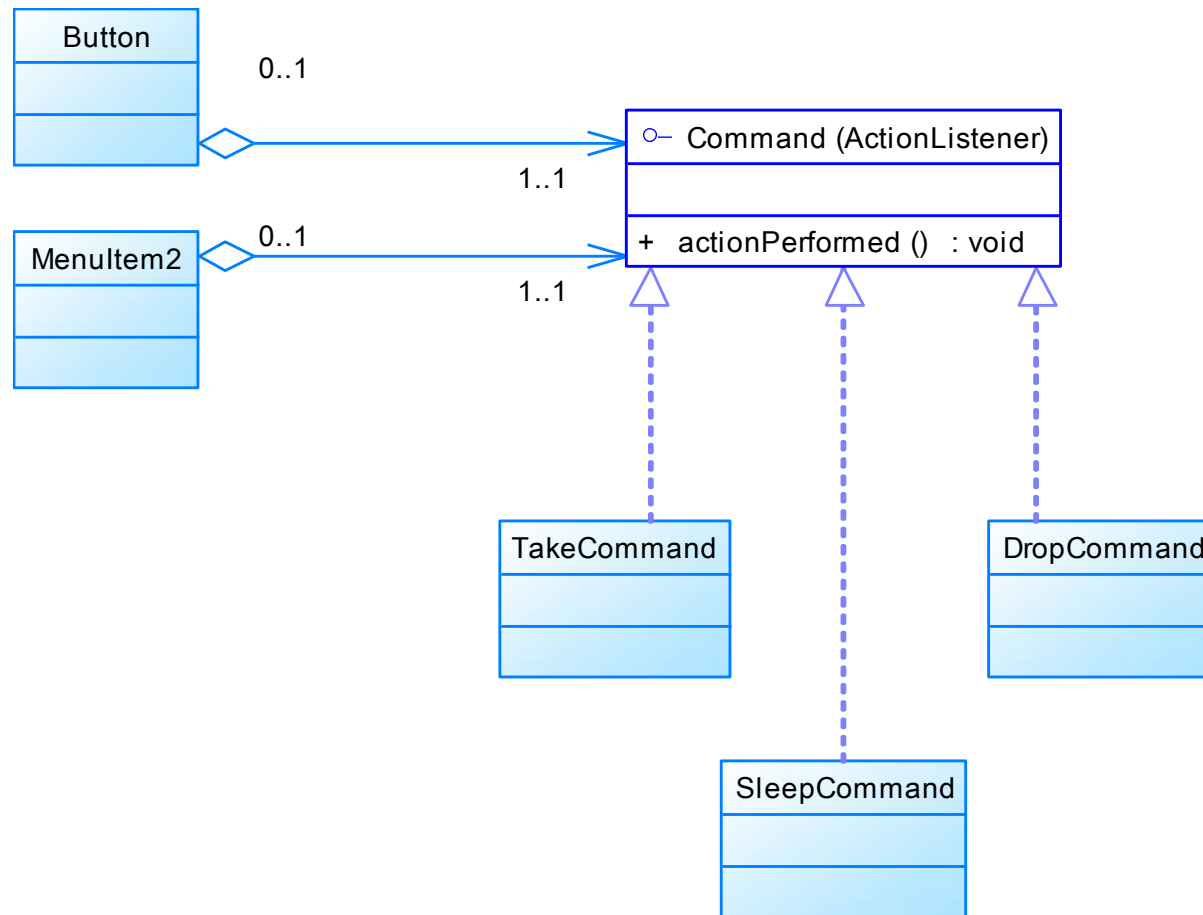
- ✓ Колко трябва да е интелигентна дадена команда? Две крайности:
  - Тъпа: просто извиква получателя на действието
  - Умна: имплементира всичко в себе си без да извиква получател.
- ✓ При прости команди, които не са обвързани с възможности за отмяна на действията и не изискват аргументи може да се използват шаблони (templates) в C++ и генетични класове (generics) в Java.

# Пример 1

---

- ✓ **Ситуация:** Система с графичен потребителски интерфейс има няколко бутона, които извършват различни действия. Искаме да има кореспондиращи опции в менюто, които да извършват същите действия.
- ✓ **Решение:** Създаване на обект-команда за всяка двойка бутон – меню-опция. Действията, които трябва да се изпълнят се имплементират в *execute* метода на обектите-команди.
- ✓ В Java за тази цел може да използваме Swing Action обекти. При тях метода се нарича *actionPerformed()*.
- ✓ <http://docs.oracle.com/javase/7/docs/api/javax/swing/Action.html>

# Пример 1



# Пример 1

```
/**
 * Пример за шаблона Команда към дисциплината "Шаблини за проектиране"
 */
package fmi.patterns.lections.command;

import java.awt.*;
import java.awt.event.ActionEvent;
import javax.swing.*;

/**
 * Демонстрира шаблона Команда, чрез използване на Swing Actions
 */
public class SwingActions extends JFrame {
    private JToolBar tb;
    private JTextArea ta;
    private JMenu fileMenu;
    private Action openAction;
    private Action closeAction;

    public SwingActions() {
        super("Swing Actions");
        setupGUI();
    }

    private void setupGUI() {
```



# Пример 1

```
}  
  
private void setupGUI() {  
    // Създава toolbar и меню.  
    tb = new JToolBar();  
    fileMenu = new JMenu("Файл");  
  
    // Създава текстовата област, в която ще се извежда резултата.  
    ta = new JTextArea(5, 30);  
    JScrollPane scrollPane = new JScrollPane(ta);  
  
    // Подреждане на съдържанието.  
    JPanel contentPane = new JPanel();  
    contentPane.setLayout(new BorderLayout());  
    contentPane.setPreferredSize(new Dimension(400, 150));  
    contentPane.add(tb, BorderLayout.NORTH);  
    contentPane.add(scrollPane, BorderLayout.CENTER);  
    setContentPane(contentPane);  
  
    // Настройка на менюто.  
    JMenuBar mb = new JMenuBar();  
    mb.add(fileMenu);  
    setJMenuBar(mb);  
  
    // Създаване на действие за "Отвори".  
    ImageIcon openIcon = new ImageIcon("open.png");  
    openAction = new AbstractAction("Отвори", openIcon) {  
        public void actionPerformed(ActionEvent e) {  
            ta.append("Действието Отвори от " + e.getActionCommand() + "\n");  
        }  
    };  
  
    // Използване на action за добавяне на бутон в toolbar-а.
```

# Пример 1

---

```
};

// Използване на action за добавяне на бутон в toolbar-а.
JButton openButton = tb.add(openAction);
openButton.setText("");
openButton.setActionCommand("Бутон Отвори");
openButton.setToolTipText("Това е бутона за отваряне");

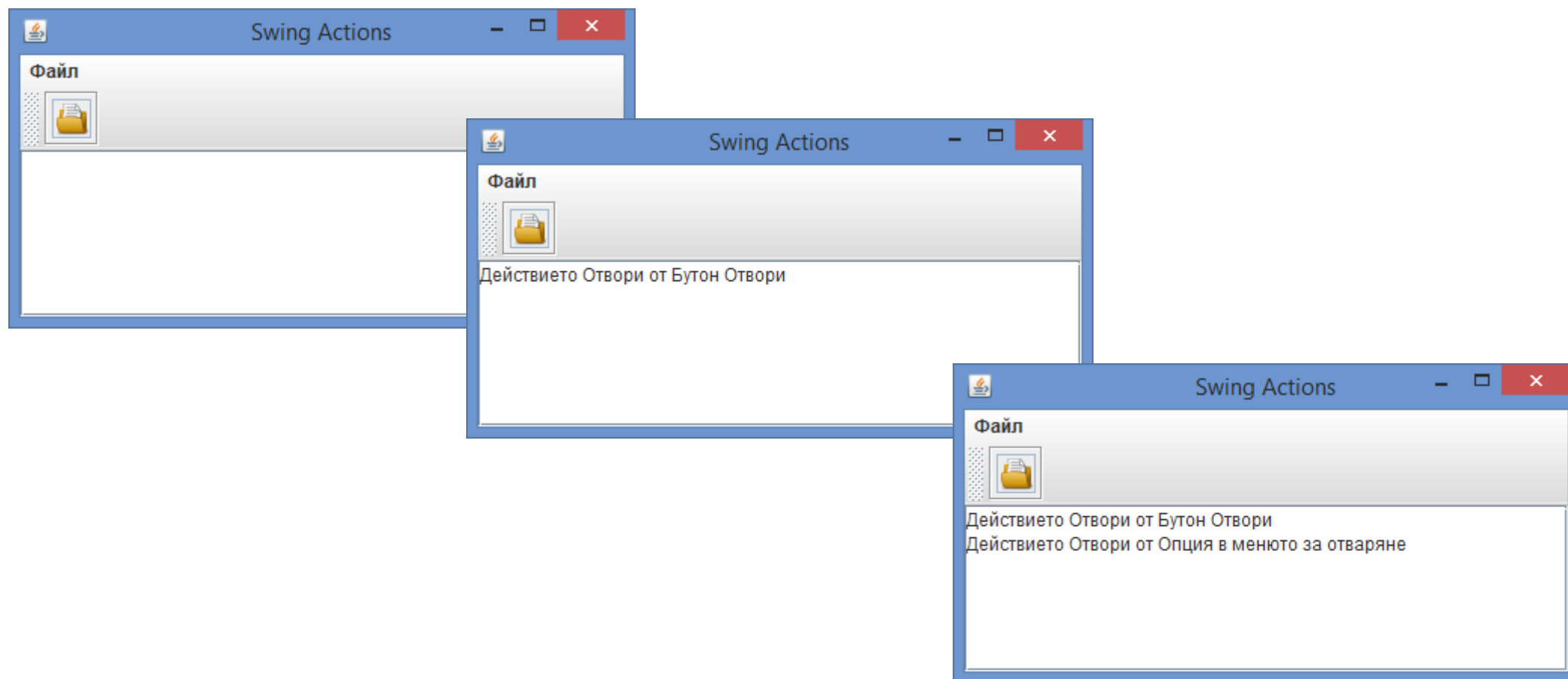
// Използване на action за добавяне на опция във менюто Файл.
JMenuItem openMenuItem = fileMenu.add(openAction);
openMenuItem.setIcon(null);
openMenuItem.setActionCommand("Опция в менюто за отваряне");

// По подобен начин се създава action за "Затваряне" и
// се използва за добавяне на бутон в toolbar-а и опция в менюто.
// Кодът е оставен за самостоятелна работа.
}

public static void main(String[] args) {
    SwingActions frame = new SwingActions();
    frame.pack();
    frame.setVisible(true);
}
}
```

# Пример 1

---



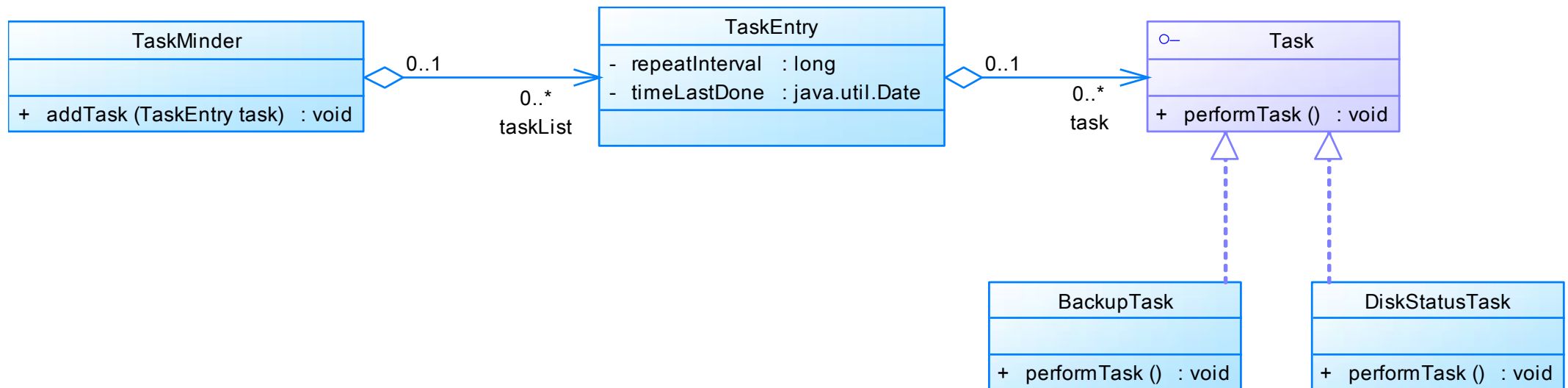
# Пример 2

---

- ✓ **Сценарий:** Искаме да направим клас, който периодично да изпълнява един или повече методи на различни обекти.
- ✓ Например искаме да стартираме backup операция на всеки час и операция за проверка на статуса на твърдия диск на всеки 10 минути.
- ✓ Не искаме класа да е обвързан с детайлите на тези операции или с обектите, които ги предоставят.
- ✓ Искаме да разделим класа, който поддържа разписанието за изпълнение на операциите от класовете, които реално имплементират поведението, което искаме да се изпълнява.

# Пример 2

✓ Решение: Шаблонът *Команда*!



# Пример 2

---

## ✓ Интерфейсът *Task*

```
/**
 * Пример за шаблона Команда към дисциплината "Шаблони за проектиране"
 */
package fmi.patterns.lections.command;

public interface Task {
    public void performTask();
}
```

# Пример 2

---

- ✓ Вместо BackupTask или DiskStatusTask създаваме прост FortuneTellerTask, който просто обхожда списък с предсказания

```
public class FortuneTask implements Task {  
  
    int nextFortune = 0;  
    String[] fortunes = { "Който учи той ще сполучи",  
                          "Опознай шаблоните и научи истината",  
                          "Сговорна дружина планина повдига" };  
  
    public FortuneTask() {  
    }  
  
    @Override  
    public void performTask() {  
        System.out.println("Твоето бъдеще е: " + fortunes[nextFortune]);  
        nextFortune = (nextFortune + 1) % fortunes.length;  
    }  
  
    public String toString() {  
        return ("Task: Предсказател");  
    }  
}
```

# Пример 2

---

- ✓ И Fibonacci Sequence Task, който генерира последователност от числа на Фибоначи.

```
public class FibonacciTask implements Task {  
    int n1 = 1;  
    int n2 = 0;  
    int num;  
  
    public FibonacciTask() {  
    }  
  
    @Override  
    public void performTask() {  
        num = n1 + n2;  
        System.out.println("Следващото число на Фибоначи е: " + num);  
        n1 = n2;  
        n2 = num;  
    }  
  
    public String toString() {  
        return ("Task: числа на Фибоначи");  
    }  
}
```



# Пример 2

## ✓ Класа *TaskEntry*

```
public class TaskEntry {  
    /**  
     * Задачата, която трябва да бъде изпълнена. Това е обект Команда!  
     */  
    private Task task;  
  
    /**  
     * Колко често трябва да бъде изпълнявана задачата  
     */  
    private long repeatInterval;  
  
    /**  
     * Кога за последно е била изпълнена задачата  
     */  
    private long timeLastDone;  
  
    public TaskEntry(Task task, long repeatInterval) {  
        this.task = task;  
        this.repeatInterval = repeatInterval;  
        this.timeLastDone = System.currentTimeMillis();  
    }  
  
    public Task getTask() {
```

## Пример 2

---

```
public Task getTask() {
    return task;
}

public void setTask(Task task) {
    this.task = task;
}

public long getRepeatInterval() {
    return repeatInterval;
}

public void setRepeatInterval(long ri) {
    this.repeatInterval = ri;
}

public long getTimeLastDone() {
    return timeLastDone;
}

public void setTimeLastDone(long t) {
    this.timeLastDone = t;
}

public String toString() {
    return (task + " трябва да бъде изпълнявана на всеки " + repeatInterval
        + " ms; за последно е изпълнена на " + timeLastDone);
}
}
```

# Пример 2

## ✓ Класът *TaskMinder*

```
public class TaskMinder extends Thread {  
    /**  
     * Колко често TaskMinder трябва да проверява за изпълнение на задачи  
     */  
    private long sleepInterval;  
  
    /**  
     * Списък със задачи  
     */  
    private ArrayList<TaskEntry> taskList;  
  
    public TaskMinder(long sleepInterval) {  
        this.sleepInterval = sleepInterval;  
        taskList = new ArrayList<TaskEntry>();  
        start();  
    }  
  
    public void addTask(Task task, long repeatInterval) {  
        long ri = (repeatInterval > 0) ? repeatInterval : 0;  
        TaskEntry te = new TaskEntry(task, ri);  
        taskList.add(te);  
    }  
  
    public Iterator<TaskEntry> getTasks() {
```

# Пример 2

---

```
}

public Iterator<TaskEntry> getTasks() {
    return taskList.iterator();
}

public long getSleepInterval() {
    return sleepInterval;
}

public void setSleepInterval(long si) {
    this.sleepInterval = si;
}

public void run() {
    while (true) {
```

# Пример 2

---

```
}  
  
public void run() {  
    while (true) {  
        try {  
            sleep(sleepInterval);  
            long now = System.currentTimeMillis();  
            Iterator<TaskEntry> it = taskList.iterator();  
            while (it.hasNext()) {  
                TaskEntry te = (TaskEntry) it.next();  
  
                if (te.getRepeatInterval() + te.getTimeLastDone() < now) {  
                    te.getTask().performTask();  
                    te.setTimeLastDone(now);  
                }  
            }  
        } catch (Exception e) {  
            System.out.println("Interrupted sleep: " + e);  
        }  
    }  
}
```

# Пример 2

---

✓ Тестова програма

```
public class TestTaskMinder {  
  
    public static void main(String[] args) {  
        // Създаване и стартиране на TaskMinder.  
        // Този TaskMinder проверява за задачи на всеки 500 ms.  
        TaskMinder tm = new TaskMinder(500);  
  
        // Създаване на задача Fortune Teller.  
        FortuneTask fortuneTask = new FortuneTask();  
  
        // Изпълнение на Fortune Teller на всеки 3 s.  
        tm.addTask(fortuneTask, 3000);  
  
        // Създаване на задача Fibonacci.  
        FibonacciTask fibonacciTask = new FibonacciTask();  
  
        // Изпълнение на задачата на всеки 700 ms.  
        tm.addTask(fibonacciTask, 700);  
    }  
}
```

# Пример 2

---

## ✓ Изход на тестовата програма

```
Следващото число на Фибоначи е: 1
Следващото число на Фибоначи е: 1
Твоето бъдеще е: Който учи той ще сполучи
Следващото число на Фибоначи е: 2
Следващото число на Фибоначи е: 3
Следващото число на Фибоначи е: 5
Твоето бъдеще е: Опознай шаблоните и научи истината
Следващото число на Фибоначи е: 8
Следващото число на Фибоначи е: 13
Следващото число на Фибоначи е: 21
Твоето бъдеще е: Сговорна дружина планина повдига
Следващото число на Фибоначи е: 34
Следващото число на Фибоначи е: 55
Следващото число на Фибоначи е: 89
Твоето бъдеще е: Който учи той ще сполучи
Следващото число на Фибоначи е: 144
Следващото число на Фибоначи е: 233
```

# Край: Шаблон Команда (Command)

---

ЛЕКЦИОНЕН КУРС: ШАБЛОНИ ЗА ПРОЕКТИРАНЕ