מרכז ההדרכה 2000
תמיכה ועדכונים
עדכון מס' 48
מאי 2002

מימוש מכונת מצבים (FSM) באמצעות State Pattern

מבוא

מכונת מצבים סופית (**Final State Machine**) היא מודל מקובל בניתוח מערכות באופן כללי, ומערכות חומרה ותוכנה בפרט. במודל זה, ליישות מסויימת נקבעים מספר מצבים בהם היא יכולה להיות, והמעברים בין המצבים מתבצעים כתלות בקלט כלשהו.

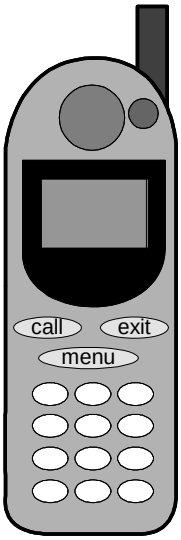
למעשה, ניתן לתאר כל עצם תוכנה כמכונת מצבים: קבוצת הנתונים שהוא כולל מגדירה את המצב בו הוא נמצא - כלומר, ערכיהם ברגע מסויים מתאר את מצבו באותו רגע. השתנות ערכים אלו כתלות בקלט או באירועים מסויימים מהווה מעבר מצב.

במאמר זה נכיר Pattern ידוע לייצוג מכונות מצבים הנקרא **State**, ונעמוד על יתרונותיו ביחס ללוגיקת משפטי תנאי (if / switch) תוך שימוש בפולימורפיזם מתקדם. להרחבה בנושא State ו- Design Patterns עיין/י בספר "***C++ - מדריך מקצועי***" בהוצאת "**מרכז ההדרכה 2000**".

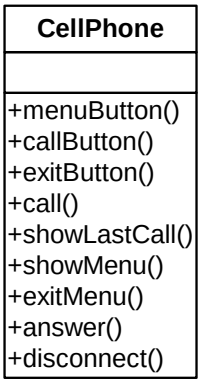
State Pattern

בעיה

נדרש לבנות תוכנה עבור מכשיר טלפון סלולרי. בטלפון זה שלושה מקשים עיקריים, מלבד מקשי הספרות/אותיות:



המכשיר הסלולרי ייוצג ע"י המחלקה **CellPhone**, שתכלול את השירותים הבאים:



<u>הסבר</u>: שלוש הפונקציות הראשונות הן פונקציות התגובה ללחיצות על שלושת הכפתורים המתאימים במכשיר:

menuButton() - נלחץ המקש "menu"

callButton() - נלחץ המקש "call"

exitButton() - נלחץ המקש "exit"

שאר הפונקציות:

call() - ביצוע התקשרות

showLastCall() - הצגת המספר האחרון שאליו התקשרנו מהמכשיר

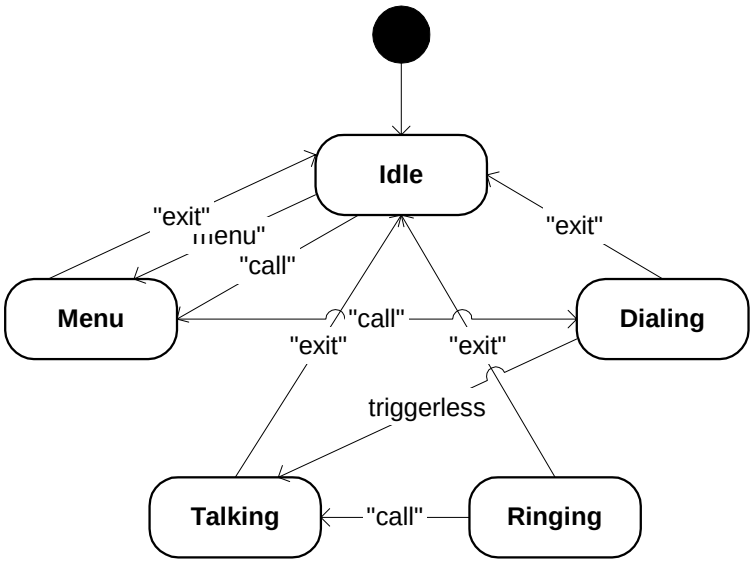
showMenu() - הצגת התפריט

exitMenu() - יציאה מהתפריט

answer() - מענה לשיחה (בזמן צלצול)

disconnect() - ניתוק השיחה.

להלן תרשים המצבים של המערכת:



- לחיצה על "menu" מציגה את התפריט. במידה והתפריט כבר מוצג לא מבוצע דבר. בזמן שיחה, צלצול וחיוג לא מבוצע דבר.
- לחיצה על "call" גורמת לחיוג למספר המוצג על המסך (קריאה ל- ()call). במידה ולא מוצג מספר כשלהו, לחיצה זו תגרום להצגת המספר האחרון אליו חייגנו (קריאה ל- ()showLastCall). במידה

והמכשיר מצלצל, לחיצה על כפתור זה תגרום למענה לשיחה (קריאה ל- ()answer).

- לחיצה על "exit" גורמת לניתוק השיחה הנוכחית (קריאה ל- ()disconnect) במידה ואנו באמצע שיחה, להפסקת הצלצול במידה והמכשיר מצלצל, או להפסקת החיוג כאשר המכשיר במצב חיוג. במצב של הצגת התפריט, לחיצה על "exit" גורמת ליציאה מהתפריט (קריאה ל- ()exitMenu).

הערה : המימוש כאן הוא פשטני ואינו כולל את הטיפול המלא בכלל השירותים של מכשיר סלולרי מודרני.

פתרון שגוי

נגדיר enum של אוסף המצבים האפשריים של המכשיר, ונבצע משפט switch-case בכל פונקצית תגובה לבדיקת המצב הנוכחי.

כך מוכרזת המחלקה CellPhone :

```
class CellPhone
{
	enum State { IDLE, DIALING, RINGING, TALKING, MENU};
	State m_state;
public:
	// Buttons input response methods
	void menuButton();
	void callButton();
	void exitButton();

	// cellular methods
	void call();
	void showLastCall();
	void showMenu();
	void exitMenu();
	void answer() ;
	void disconnect();
};
```

פונקצית התגובה ללחיצה על "menu" :

```
void CellPhone::menuButton()
{
	switch(m_state)
	{
	case IDLE:
		m_state = MENU;
		showMenu();
		break;

	case DIALING:
	case RINGING:
	case TALKING:
	case MENU:
		break;		// do nothing
	}
}
```

פונקצית התגובה ללחיצה על "call" :

```
void CellPhone::callButton()
```

```
{
        switch(m_state)
        {
        case IDLE:
                m_state = MENU;
                showLastCall();
                break;

        case RINGING:
                m_state = TALKING;
                answer();
                break;

        case MENU:
                m_state = DIALING;
                call();
                m_state = TALKING;
                break;

        case DIALING:
        case TALKING:
                break;                // do nothing
        }
}
```

ופונקצית התגובה ללחיצה על "exit":

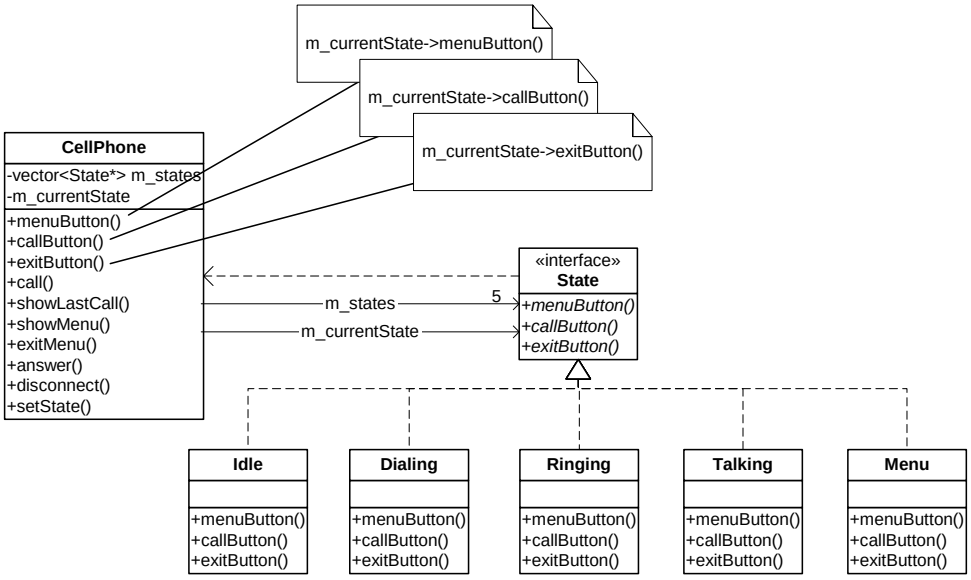
```
void CellPhone::exitButton()
{
        switch(m_state)
        {
        case RINGING:
        case DIALING:
        case TALKING:
                disconnect();
                m_state = IDLE;
                break;

        case MENU:
                exitMenu();
                m_state = IDLE;
                break;

        case IDLE:
                break;                // do nothing
        }
}
```

חסרונות הפתרון:

- **יעילות**: חיפוש המצב הנוכחי במשפט switch-case אינו יעיל, מכיוון שבאופן ממוצע, עוברים בכל קריאה לפונקציה כנ"ל על מחצית מספר המצבים. כמו כן משפט switch-case אינו שומר את המידע שנרכש (מציאת המצב המתאים) עבור הפונקציות הבאות, גם כאשר המצב לא שונה.
- **מודולריות**: קוד המחלקה CellPhone אינו מודולרי. הטיפול בכל המצבים מבוצע באופן ריכוזי במשפט switch-case במחלקה, ומחייב שינויים במספר מקומות בכל שינוי או הוספה בדרישות היישום.

פתרון: State Pattern

רעיון הפתרון הוא להשתמש בפולימורפיזם כתחליף למשפט switch-case: נגדיר מחלקת מצב אבסטרקטית, **State**, ומחלקות נורשות ממנה המייצגות את מצבי המערכת השונים:



- המחלקה CellPhone מחזיקה וקטור של מצביעים ל- State (m_states) ומצביע נוסף ל- State הנוכחי (m_currentState).
- State מחזיקה גם היא מצביע למחלקה CellPhone.
- לכל מצב מוגדרת מחלקה מתאימה הנורשת מ- State וממשת את שלוש פונקציות התגובה ללחיצה על המקשים.
- כאשר נלחץ המקש "menu" לדוגמא, מופעלת הפונקציה CellPhone::menuButton(), וזו מצבעת הפנייה (Forwarding) לפונקציה m_currentState->menuButton(), כלומר, לפונקציה של המצב הנוכחי.
- הפונקציות המפנות במחלקה CellPhone הן **inline**, ולכן אין פה אובדן יעילות בביצוע קריאות עקיפות לפונקציות של State.

כך מוכרז הממשק **State**:

```
class State
{
public:
        virtual void menuButton(CellPhone *cellPhone) = 0;
        virtual void callButton(CellPhone *cellPhone) = 0;
        virtual void exitButton(CellPhone *cellPhone) = 0;
};
```

וכך למשל מוגדרת המחלקה Idle היורשת מ- State ומייצג מצב המתנה:

```
class Idle : public State
{
public:
        void menuButton(CellPhone *cellPhone)
        {
                cellPhone->showMenu();
                cellPhone->setState(MENU);
        }
        void callButton(CellPhone *cellPhone)
        {
                cellPhone->showLastCall();
```

```
                cellPhone->setState(MENU);
            }
            void exitButton(CellPhone *cellPhone) {}
        };
```

הסבר: Idle מממשת את הפונקציות הוירטואליות הטהורות שהוגדרו ב- State, בהתאם למוגדר עבור מצב IDLE.

המחלקה הראשית, CellPhone, מבצע הפנייה (forwarding) של קריאות לפונקציות מצב למחלקת המצב הנוכחי:

```
class CellPhone
{
public:
        enum State_enum {IDLE, DIALING, RINGING, TALKING, MENU, STATE_SIZE};

        CellPhone() : m_states(STATE_SIZE)
        {
                m_states.push_back(new Idle);
                m_states.push_back(new Dialing);
                m_states.push_back(new Ringing);
                m_states.push_back(new Talking);
                m_states.push_back(new Menu);
                m_currentState = m_states[IDLE];
        }

        void setState(State_enum s)      { m_currentState = m_states[s]; }

        // Buttons input response methods
        void menuButton()
        { m_currentState->menuButton(this); }
        void callButton()
        { m_currentState->callButton(this); }
        void exitButton()
        { m_currentState->exitButton(this); }

        // cellular methods
        void call();
        void showLastCall();
        void showMenu();
        void exitMenu();
        void answer() ;
        void disconnect();

private:
        State*              m_currentState;
        vector<State*>      m_states;
};
```

שיפור נוסף : קינון מחלקות והסתרת מידע

מכיוון שהמשתמש במחלקה אינו אמור לעשות שימוש במחלקות המצב, כדאי להגדירן כמקוננות private בתוך המחלקה CellPhone.

הקוד המלא של המחלקה נראה אם כן כעת כך :

```
#include <vector>
using namespace std;

class CellPhone
{
public:
	enum State_enum {IDLE, DIALING, RINGING, TALKING, MENU, STATE_SIZE};

	CellPhone() : m_states(STATE_SIZE)
	{
		m_states.push_back(new Idle);
		m_states.push_back(new Dialing);
		m_states.push_back(new Ringing);
		m_states.push_back(new Talking);
		m_states.push_back(new Menu);
		m_currentState = m_states[IDLE];
	}

	void setState(State_enum s) { m_currentState = m_states[s]; }

	// Buttons input response methods
	void menuButton()
	{ m_currentState->menuButton(this);}
	void callButton()
	{ m_currentState->callButton(this);}
	void exitButton()
	{ m_currentState->exitButton(this);}

	// cellular methods
	void call();
	void showLastCall();
	void showMenu();
	void exitMenu();
	void answer() ;
	void disconnect();


private:

	// state interface
	class State
	{
	public:
		virtual void menuButton(CellPhone *cellPhone) = 0;
		virtual void callButton(CellPhone *cellPhone) = 0;
		virtual void exitButton(CellPhone *cellPhone) = 0;
	};

	class Idle : public State
	{
	public:
		void menuButton(CellPhone *cellPhone)
		{
			cellPhone->showMenu();
			cellPhone->setState(MENU);
```

```
        }
        void callButton(CellPhone *cellPhone)
        {
                cellPhone->showLastCall();
                cellPhone->setState(MENU);
        }
        void exitButton(CellPhone *cellPhone) {}
};

class Dialing: public State
{
public:
        void menuButton(CellPhone *cellPhone) {}
        void callButton(CellPhone *cellPhone) {}
        void exitButton(CellPhone *cellPhone)
        {
                cellPhone->disconnect();
                cellPhone->setState(IDLE);
        }
};
class Ringing: public State
{
public:
        void menuButton(CellPhone *cellPhone)   {}
        void callButton(CellPhone *cellPhone)
        {
                cellPhone->answer();
                cellPhone->setState(TALKING);
        }
        void exitButton(CellPhone *cellPhone)
        {
                cellPhone->disconnect();
                cellPhone->setState(IDLE);
        }
};

class Talking: public State
{
public:
        void menuButton(CellPhone *cellPhone) {}
        void callButton(CellPhone *cellPhone) {}
        void exitButton(CellPhone *cellPhone)
        {
                cellPhone->disconnect();
                cellPhone->setState(IDLE);
        }
};

class Menu : public State
{
public:
        void menuButton(CellPhone *cellPhone) {}
        void callButton(CellPhone *cellPhone)
        {
                cellPhone->call();
                cellPhone->setState(DIALING);
        }
        void exitButton(CellPhone *cellPhone)
        {
                cellPhone->disconnect();
                cellPhone->setState(IDLE);
        }
```

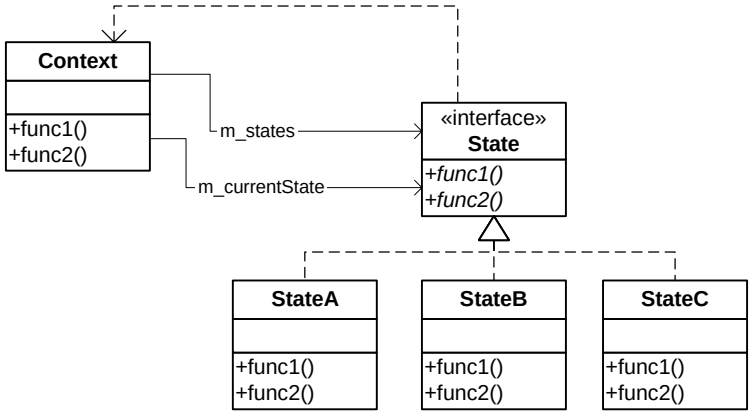
```
    };

    State*          m_currentState;
    vector<State*>  m_states;
};
```

כפי שניתן לראות, פתרון זה עדיף משתי הבחינות שהוזכרו על פני הפתרון הקודם:

- **יעילות**: המצב הנוכחי נשמר, ולא מבוצע חיפוש כלשהו. במלים אחרות, לעומת סיבוכיות חיפוש המצב בפתרון הקודם O(n), הסיבוכיות כאן היא O(1).
- **מודולריות**: הפתרון מודולרי יותר עקב המבנה הפולימורפי. הוספה של מצב חדש מתבטאת בהוספת מחלקה חדשה, ובהוספת עצם ממנה באתחול. כמו כן שינוי בהתנהגות במצב מסוים משפיעה אך ורק על הקוד של מחלקת המצב המתאימה.

הכללה

State הוא Pattern המשמש להגדרת מכונת מצבים:



לכל מצב של מחלקה נתונה (context) מגדירים מחלקה היורשת מהממשק State וממשת את הפונקציות המוכרזות בו. פונקציות אלו מאופיינות בכך שביצוען תלוי במצב של Context.

- **אפשרויות ווריאציות**:
  - ה- Context יכול ליצור את וקטור המצבים האפשריים ולהקצות אותם מראש, או באופן הדרגתי, לייצר עצם בהגעה למצב המתאים.
  - שינוי מצב של המערכת: יכול להתבצע ע"י ה- Context, ע"י ה- States או ע"י שניהם.

סיכום

מימוש מכונת מצבים (FSM) במערכות מונחות עצמים ניתן לביצוע ע"י ה- State Pattern. למימוש זה יתרונות בולטים על פני המימוש המסורתי ע"י משפט Switch:

- הוא יעיל יותר
- הוא מודולרי יותר

כאשר מספר המצבים רב ו/או קיימת היררכייה של מכונות מצבים (תת מכונות מצבים), המימוש ע"י State Pattern מסורבל ומורכב. במצב זה, פתרון פשוט יותר הוא מימוש ע"י טבלת מעברי מצב.

כל הזכויות שמורות © מאיר סלע

מרכז ההדרכה 2000