Model View Control Generic

Vlad C. Manea
anul al III-lea, grupa 6
vlad.manea@info.uaic.ro

---

1. Problemă

Să se proiecteze, implementeze și exemplifice o aplicație care ilustrează modul de funcționare al șablonului de proiectare **ModelViewControl** (MVC).

---

2. Soluție

Deoarece nu este specificat exemplul pe care trebuie să funcționeze produsul final, am modelat aplicația ca fiind una generică, doar inspirându-mă din exemplul de la laborator, dar dezvoltând totul separat de acesta. Am ales această variantă mai consumatoare de timp deoarece am avut în vedere ca aplicația mea să nu se rezume la un studiu de caz fixat, ci să permită orice studiu de caz.

Am împărțit codul în interfețe și clase, astfel încât să fie specificat fiecare nivel de abstractizare. Am avut în vedere, de la abstract la concret, nivelurile:

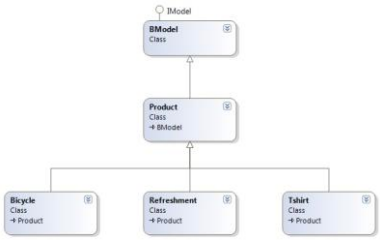
- toate modelele/viewurile/controlurile (interfață + clasă de bază);
- doar modelele/controlurile de tip „produs” (clasă);
- doar modelele/viewurile/controlurile de tip „tip de produs” (clasă).

Același lucru se întâmplă la metode. Dacă în exemplul de la laborator, în interfața pentru model apar metode clare (ex: `Turn`, `Accelerate`, `Decelerate`), în interfața mea pentru modele, `IModel`, apare doar un operator `this[]`, care permite dinamic specificarea de metode după nume la runtime.
Pe aceeași idee, în control și view, acțiunile specifice respectivului element sunt modelate cu operatorul `this[]`, oferind maximă flexibilitate în utilizare.

Studiul de caz este modelarea unor produse într-un magazin universal. Acestea sunt încadrate în aplicația MVC generică construită. Diagramele de clase și interfețe se găsesc în directorul proiectului în directorul `Diagrame`. Ea conține deopotrivă MVC generic și studiul de caz.

---

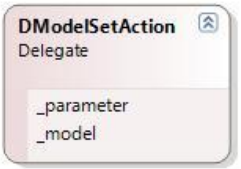
3. Model



Modelul este compus din interfața `IModel`, clasa de bază `BModel` care implementează interfața. Clasele care derivează din `BModel` fac parte din studiul de caz, și anume modelarea unor produse oarecare. Mai jos se găsesc produse de tipuri concrete, modelate prin clase: o bicicletă (`Bicycle`), o băutură răcoritoare (`Refreshment`) și un tricou (`Tshirt`).

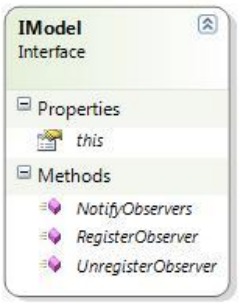
Acțiunile de get/set asupra datelor adăugate dinamic în model sunt realizate cu ajutorul delegaților transmiși modelului împreună cu numele proprietății pe care o modifică/preiau și valoarea ei.

Delegatul de set, specificat în spațiul de nume pentru modele, are formatul din figura alăturată. Astfel, toți delegații de set se vor comporta uniform. Același lucru se petrece cu delegații de get.

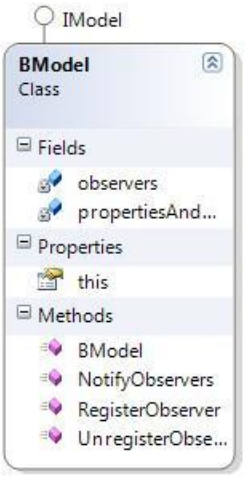


Interfața `IModel` cere tuturor modelelor să se conformeze cu șablonul de proiectare Observer: viewurile sunt observatorii, iar modelul este observatul. Metodele standard pentru aceasta sunt:

- `NotifyObservers` – parcurge toți observatorii și îi notifică la actualizarea datelor din model,
- `RegisterObserver` – înregistrează un observator,
- `UnregisterObserver` – elimină un observator.



Clasa de bază `BModel` implementează interfața `IModel` și conține un `HashSet` de observatori și un `Dictionary` de proprietăți și delegați de forma `<nume_proprietate:String, tuplu<conținut:Object, delegat_get:Delegat, delegat_set:Delegat>>`, prescurtată **DM**.
Cu alte cuvinte, modelul va ști la runtime că proprietatea cu numele `nume_proprietate` are valoarea reținută în `conținut` și poate fi accesată prin delegatul `delegat_get` și modificată prin delegatul `delegat_set`.



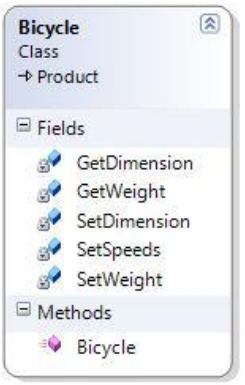
Pentru simplitate, am folosit operatorul `this[]`, cu semnificația că apelul `this[nume_proprietate, indirect] = valoare` duce la executarea delegatului de set pentru respectiva proprietate care primește ca parametru `valoare`, iar `variabilă = this[nume_proprietate, indirect]` duce la executarea delegatului de get pentru respectiva proprietate. Am implementat și varianta `direct`, în care se setează sau obține direct obiectul `valoare` asociat proprietății. Nu este obligatorie specificarea de delegați. Dacă un astfel de delegat nu este specificat, variabila este modificată sau obținută direct.

Clasa `Product` derivează din `BModel` și conține nimic altceva decât o serie de delegați specifici tuturor produselor, și anume `GetName`, `SetName`, `GetPrice`, `SetPrice`, `SetStock`. Delegații lipsă duc la operații directe care îi înlocuiesc.

Vom studia în amănunt partea studiului de caz ce ține de biciclete. Celelalte două tipuri de produse au proprietăți similare. Clasa `Bicycle` derivează din `Product` și conține delegații specifici proprietăților unei biciclete: dimensiune, greutate, număr de viteze.

Constructorul clasei primește un dicționar de tipul **DM**, adaugă la acesta propriile nume de proprietăți cu delegații asociați și trimite dicționarul modificat spre clasa model părinte (`Product`).

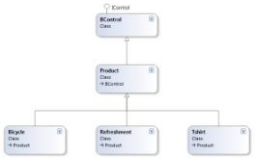
Aceasta procedează la fel și trimite dicționarul (care acum conține proprietăți și modalități de gestiune și de la produse, și de la biciclete) clasei de bază (`BModel`), care reține tot în dicționarul `propertiesAndActions`.



---

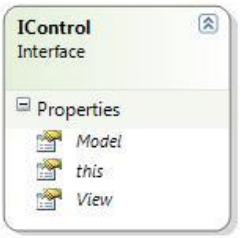
4. Control

Controlul respectă o structură similară cu modelul în ceea ce privește separarea nivelelor de abstractizare. Rolul controlului în aplicație este să execute acțiuni asupra modelului la comanda viewului. Aplicația permite și aici specificarea

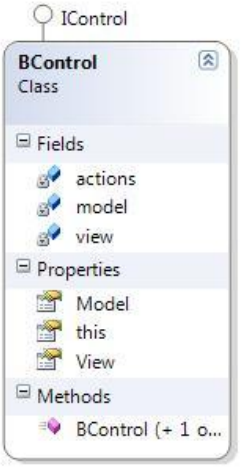


acțiunilor în mod dinamic. Interfața `IControl` este implementată de `BControl`, derivată la rândul ei de clasa `Product` și, mai departe, de produse concrete.

Interfața `IControl` specifică necesitatea prezenței viewului și a modelului pentru toate controlurile.



Clasa de bază `BControl` implementează interfața `IControl` și conține, pe lângă referințele spre model și view, un `Dictionary` de acțiuni de forma `<nume_acțiune:String, delegat_acțiune:Delegat>>`, prescurtată **DC**.
Cu alte cuvinte, modelul va ști la runtime că acțiunea cu numele `nume_acțiune` duce la executarea delegatului `delegat_acțiune`.



Pentru simplitate, am folosit operatorul `this[]`, cu semnificația că apelul `this[nume_acțiune] = valoare` duce la executarea delegatului de acțiune pentru respectiva proprietate care primește ca parametru `valoare`. În acest caz, toate acțiunile sunt specificate.

Modelarea cu acțiuni permite orice business logic pentru orice studiu de caz. Bineînțeles, dacă nu există delegați care să gestioneze business logica pentru anumite date din model, acestea nu vor fi gestionate prin control

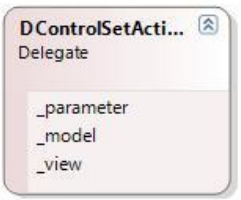
Clasa `Product` derivează din `BModel` și conține nimic altceva decât o serie de delegați specifici acțiunilor asupra tuturor produselor, și anume `IncDecPrice` și `IncDecStock` (pentru incrementarea/decrementarea prețului, respectiv stocului) `ModName` și `ModProducer` (pentru modificarea numelui și a producătorului).

Clasa `Bicycle` derivează din `Product` și conține delegații specifici acțiunilor asupra proprietăților unei biciclete: dimensiune, greutate, număr de viteze.

Constructorul clasei primește un dicționar de tipul **DC**, adaugă la acesta propriile nume de proprietăți cu delegații asociați și trimite dicționarul modificat spre clasa control părinte (`Product`).

Aceasta procedează la fel și trimite dicționarul (care acum conține proprietăți și modalități de acționare asupra datelor și de la produse, și de la biciclete) clasei de bază (`BControl`), care reține tot în dicționarul `actions`.

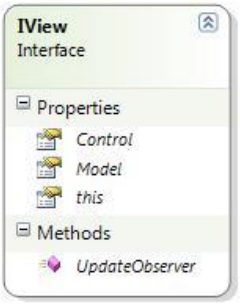
Delegatul de actiune, specificat în spațiul de nume pentru control, are formatul din figura alăturată. Astfel, toți delegații de acțiune se vor comporta uniform.



4. View

Viewul respectă o structură ușor diferită de control în ceea ce privește separarea nivelelor de abstractizare. Rolul viewului în aplicație este să transmită intenții de modificare spre control și să fie notificat de model la actualizarea acestuia din urmă. În plus, viewul este dator cu actualizarea dictată de control: aplicația permite și aici specificarea acțiunilor posibile din view în mod dinamic. În acest caz, există doar interfața `IView`, iar viewurile concrete implementează această interfață și derivează clasa `UserControl` pentur a afișa în mod grafic datele.

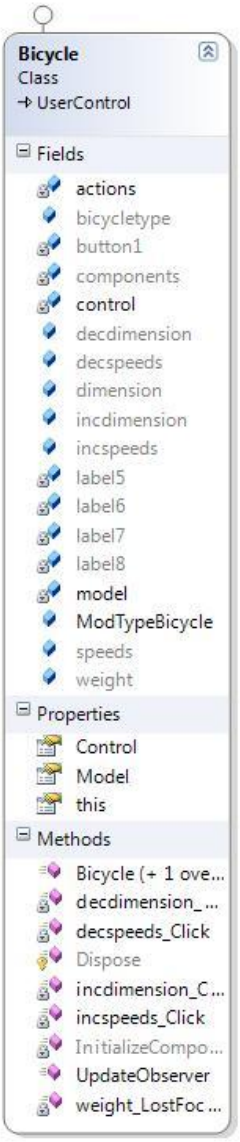
Interfața `IView` specifică necesitatea prezenței controlului și a modelului pentru toate controlurile. De asemenea, ea cere tuturor viewurilor să se conformeze cu șablonul de proiectare Observer: viewurile sunt observatorii, iar modelul este observatul.
Metoda standard pentru aceasta este `UpdateObserver`, care actualizează interfața grafică și este apelată în metoda `NotifyObservers` din model.



Voi exemplifica clasa concretă `Bicycle` care implementează interfața `IView` și derivează `UserControl`.

Ea conține:

- un dicționar cu acțiunile de actualizare (delegați) care pot fi apelate din control. Spre exemplu, la `Bicycle`, se încadrează aceasta într-un tip anume: dimensiunile mici duc la tipul *Kids*. La viewul `Refreshment`, o cantitate peste 2L duce printr-o astfel de acțiune la atenționarea *cancer*. La `Product`, prețul încadrează produsul în *scump*, *ieftin* sau *acceptabil*;
- elemente de interfață grafică, deoarece derivează din `UserControl`;
- evenimente ale elementelor de interfață grafică;
- implementarea efectivă a metodei `UpdateObserver` cu referire la interfața grafică.

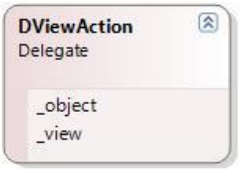


Dicționarul are forma `<nume_acțiune:String, delegat_acțiune:Delegat>>`, prescurtată **DV**. Viewul va ști la runtime că acțiunea cu numele `nume_acțiune` apelată de control duce la executarea delegatului `delegat_acțiune`.

Pentru simplitate, am folosit operatorul `this[]`, cu semnificația că apelul `this[nume_acțiune] = valoare` duce la executarea respectului delegat de acțiune, care primește ca parametru `valoare`. În acest caz, toate acțiunile sunt specificate.

Modelarea cu acțiuni permite orice actualizare de interfață pentru orice studiu de caz. Bineînțeles, dacă nu există delegați care să gestioneze interfața, pentru anumite date din model, elementele de interfață vor fi actualizate doar de model.

Delegatul de actiune, specificat în spațiul de nume pentru view, are formatul din figura alăturată. Astfel, toți delegații de acțiune se vor comporta uniform.



5. Organizare

Sursele sunt organizate astfel încât să se facă distincție între Model, View și Control:

Control	Spațiul de nume pentru controluri
BaseClasses	Spațiul de nume pentru clasele control de bază
BControl	Clasa de bază pentru controluri
ConcreteClasses	Spațiul de nume pentru clasele control concrete
Bicycle	Clasa control concretă bicicletă
Product	Clasa control concretă produs
Refreshment	Clasa control concretă răcoritoare
Tshirt	Clasa control concretă tricou
Interfaces	Spațiul de nume pentru interfețele controlurilor
IControl	Interfața pentru controluri
Model	Spațiul de nume pentru modele
BaseClasses	Spațiul de nume pentru clasele model de bază
BModel	Clasa de bază pentru modele
ConcreteClasses	Spațiul de nume pentru clasele model concrete
Bicycle	Clasa model concretă bicicletă
Product	Clasa model concretă produs
Refreshment	Clasa model concretă răcoritoare
Tshirt	Clasa model concretă tricou
Interfaces	Spațiul de nume pentru interfețele modelelor
IModel	Interfața pentru modele
View	Spațiul de nume pentru viewuri
ConcreteComponents	Spațiul de nume pentru componentele view concrete
Bicycle	Clasa view concretă bicicletă
Product	Clasa view concretă produs
Refreshment	Clasa view concretă răcoritoare
Tshirt	Clasa view concretă tricou
Interfaces	Spațiul de nume pentru interfețele viewurilor
IView	Interfața pentru viewuri

6. Bibliografie

Model View Control – M. Cochran
http://www.c-sharpcorner.com/uploadfile/rmcochran/mvc_intro12122005162329pm/mvc_intro.aspx

Proprietăți dinamice – Stack Overflow
http://stackoverflow.com/questions/947241/c-dynamic-properties