Modele de PROIECTARE

Tudor Orha

Modelele de proiectare constituie un element foarte important în proiectarea aplicațiilor software. În această serie de articole vom prezenta câteva modele de design utile în dezvoltarea sistemelor informatice.

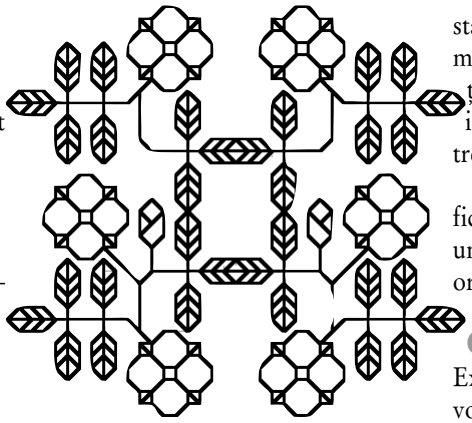
Ce este un model de design?

Orice model descrie o problemă ce apare în mod repetat în mediul nostru de lucru, după care descrie principiul soluției respectivei probleme, într-un asemenea fel încât să poată fi refolosit de milioane de ori, scrie *Cristopher Alexander* în *A Pattern Language*.

Această definiție este valabilă în orice domeniu, atât în proiectarea și construcția de clădiri sau avioane, cât și în proiectarea și construcția de *software*.

Elementele constitutive ale unui model sunt prezentate în continuare:

- ***Numele*** modelului reprezintă o modalitate de a descrie o problemă de design, soluțiile și consecințele lor într-un cuvânt sau două. Aceasta ajută la formarea unui vocabular comun, ceea ce conduce la o mai ușoară comunicare cu alte persoane.
- ***Problema*** descrie situațiile în care se poate aplica un model. Se explică problema și contextul ei. Uneori, se pot descrie probleme specifice; de exemplu se poate arăta cum pot fi reprezentați algoritmii ca obiecte. Alteori se pot descrie structuri de clase și obiecte care sunt caracteristice unui *design* inflexibil.
- ***Soluția*** descrie elementele care compun modelul, relațiile dintre ele, responsabilitățile și modul în care elementele colaborează. Soluția nu va descrie un model concret sau o implementare, pentru că un model este un șablon care poate fi aplicat în multe situații diferite. Modelul va furniza o descriere abstractă a unei probleme de design și cum poate fi aceasta rezolvată aranjând diferitele elemente într-un anumit mod.



- ***Consecințele*** sunt rezultatul și costurile aplicării unui model. Ele sunt critice în evaluarea modelelor alternative și în înțelegerea costurilor și beneficiilor aplicării modelului. De multe ori consecințele în cazul *software*-ului sunt legate de compromisuri de spațiu și timp; ele pot fi legate de limbajul în care se realizează implementarea. Datorită faptului că reutilizarea este de multe ori un factor în designul orientat pe obiecte, consecințele unui model includ impactul lui asupra flexibilității, extensibilității sau portabilității sistemului. Dezvăluirea acestor consecințe va ajuta la înțelegerea și evaluarea lor.

Un model de design numește, abstractizează și identifică aspectele cheie ale unei structuri de modelare care îl face util în crearea unui model orientat obiect reutilizabil.

Modelul identifică clasele și instanțele participante, rolul acestora și modul în care colaborează, distribuția responsabilităților între clase și instanțe. Fiecare model se concentrează pe o problemă particulară.

Ca limbaj în care vom exemplifica modelele am ales C++, ca fiind unul dintre cele mai populare limbaje orientate pe obiecte.

Clasificarea modelelor

Există multe modele care au fost dezvoltate de-a lungul timpului și doar câteva dintre ele vor fi prezentate în acest serial. Acestea sunt clasificate în funcție de scopul lor.

Pe baza acestui criteriu, avem modele:

- ***creaționale***: descriu procesul creării obiectelor;
- ***structurale***: descriu compoziția obiectelor;
- ***comportamentale***: descriu modul în care sunt împărțite responsabilitățile între obiecte.

Modul de prezentare a modelelor

Modelele vor fi prezentate folosind un format consistent. Fiecare model

va fi impărțit în secțiuni conform șablonului următor. Astfel vom avea o prezentare uniformă a informației care va face modelele mai ușor de învățat, comparat și utilizat.

Numele și clasificarea
Numele modelului exprimă succint esența modelului. Un nume bun este esențial deoarece va intra în vocabularul proiectantului. Clasificarea reflectă schema de clasificare a modelelor introdusă anterior. Numele va fi prezentat în limba engleză.

Intenția
Este o afirmație scurtă care răspunde următoarelor întrebări:
- *Ce face modelul?*
- *Care este rațiunea și scopul lui?*
- *Ce problemă de design adresează?*

Alte denumiri
Majoritatea modelelor sunt cunoscute și sub alte denumiri decât cea utilizată cel mai frecvent.

Motivație
Este prezentat un scenariu care ilustrează o problemă de *design* și cum sunt utilizate structurile de clase și obiecte din model pentru a rezolva problema. Acest scenariu va fi util în înțelegerea descrierii mai abstracte care va urma.

Aplicabilitate
Sunt descrise situațiile în care poate fi aplicat modelul; de asemenea, în această secțiune se pot exemplifica *design*-uri neperformante. Scopul acestei secțiuni este descrierea modalității în care pot fi recunoscute aceste situații.

Structura
Structura conține o reprezentare grafică a claselor, colaborărilor și interacțiunilor din model folosind o reprezentare bazată pe *UML*. Nu vom prezenta în această serie de articole această notație.

Participanți
Sunt descrise clasele și/sau obiectele care participă în model și care sunt responsabilitățile fiecărui element prezentat.

Colaborări
Este descris modul în care participanții colaborează pentru a-și îndeplini responsabilitățile.

Consecințe
Este prezentat modul în care modelul îndeplinește obiectivul său, care sunt compromisurile și rezultatele folosirii modelului și care sunt părțile din structura sistemului care vor putea varia independent.

Implementare
Sunt prezentate probleme, sugestii sau tehnici care ar trebui să fie cunoscute în momentul implementării modelului.

Exemple
Sunt prezentate fragmente de cod care ilustrează cum s-ar putea implementa modelul în limbajul *C++*.

Modelul Builder
Vom începe prezentarea modelelor de proiectare cu unul dintre cele mai simple: *Builder*.

Intenția
Principalul obiectiv al acestui model este să separe construcția unui obiect complex de reprezentarea sa, astfel încât același proces de construcție să poată fi folosit pentru crearea mai multor reprezentări.

Motivație
O aplicație pentru citirea documentelor în format ***RTF*** (***R**ich **T**ext **F**ormat*) ar putea fi capabilă să convertească documentele *RTF* în diverse alte formate.

De exemplu, ar putea să convertească documentul în text simplu (fără formatări) sau în format ***PDF***. Problema este că lista de formate în care se poate face conversia este deschisă. Așadar, ar trebui să fie ușor de adăugat un nou format fără a modifica operația de citire.

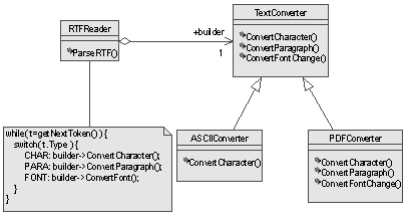
O soluție este să se configureze clasa `RTFReader` cu un obiect `TextConverter` care convertește formatul *RTF* într-o altă reprezentare a textului.

Pe măsură ce `RTFReader`-ul parcurge documentul, se folosește de `TextConverter` pentru a executa conversia.

De fiecare dată când `RTFReader` găsește un element *RTF* (fie că este text sau element de control), va emite câte o cerere către `TextConverter` ca acesta să convertească elementul.

Obiectele `TextConverter` sunt responsabile să execute conversia datelor și să reprezinte elementul în formatul corespunzător.

Subclasele clasei `TextConverter` sunt specializate pentru diferite formate și conversii. De exemplu, un `ASCIIConverter` va ignora orice cerere diferită de conversia de text, iar un `PDFConverter` va interpreta toate mesajele pentru a produce un document *PDF* care conține toate elementele stilistice prezente în documentul *RTF*.



Fiecare clasă convertor preia construcția unui obiect complex și o ascunde în spatele unei interfețe abstracte.

Convertorul (constructorul) este separat de componenta de citire, a cărei responsabilitate este de a citi și interpreta un document *RTF*.

Fiecare clasă convertor se numește *builder* în acest model, iar clasa `RTFReader` este denumită *director*.

Aplicat acestui exemplu, modelul *Builder* separă algoritmul de interpretare a unui format text de modul în care este creat și reprezentat un format convertit.

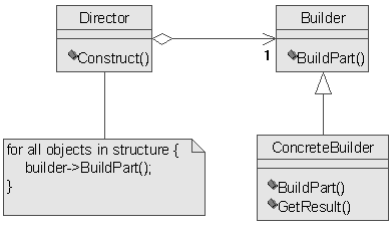
Aceasta permite refolosirea clasei `RTFReader` pentru a crea diferite reprezentări text din documente *RTF*, doar prin configurarea sa cu diferite subclase ale clasei `TextConverter`.

Aplicabilitate

Modelul *Builder* se poate folosi în următoarele situații:

- algoritmul pentru crearea unui obiect complex trebuie să fie independent de părțile care compun obiectul și felul în care ele sunt asamblate.
- procesul de construcție trebuie să permită crearea de reprezentări diferite ale obiectului care trebuie construit.

Structura



Participanți

- *Builder* (`TextConverter`):
  - ♦ definește o interfață abstractă pentru crearea părților unui obiect *Produs*;
- *ConcreteBuilder* (`ASCIIConverter`, `PDFConverter`):
  - ♦ construiește și asamblează părți ale produsului prin implementarea interfeței *Builder*;
  - ♦ definește și menține o reprezentare a produsului;
  - ♦ furnizează o interfață pentru a obține produsul (obiectul final rezultat în urma construcției);
- *Director* (`RTFReader`):
  - ♦ construiește un obiect cu ajutorul interfeței *Builder*;
- *Produsul* (text *ASCII*, document *PDF*):
  - ♦ reprezintă obiectul complex aflat în construcție. *ConcreteBuilder* construiește reprezentarea internă a produsului și definește procesul prin care este asamblat;
  - ♦ include clasele care definesc părțile constitutive, inclusiv interfețele pentru asamblarea produsului final.

Colaborări

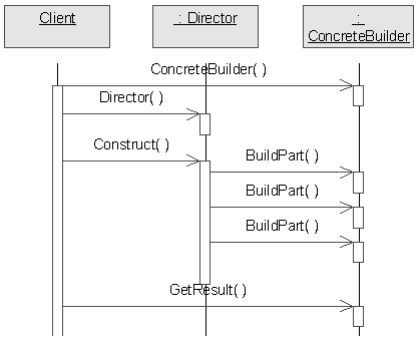
Clientul crează obiectul *Director* și îl configurează cu obiectul *Builder* dorit.

Obiectul *Director* notifică *Builder*-ul de fiecare dată când o parte a *Produsului* trebuie construită.

Builder-ul prelucrează cererile de la *Director* și adaugă părți la *Produs*.

Clientul recuperează *Produsul* de la *Builder*.

Următoarea diagramă ilustrează cum *Builder*-ul și *Director*-ul cooperează cu un client.



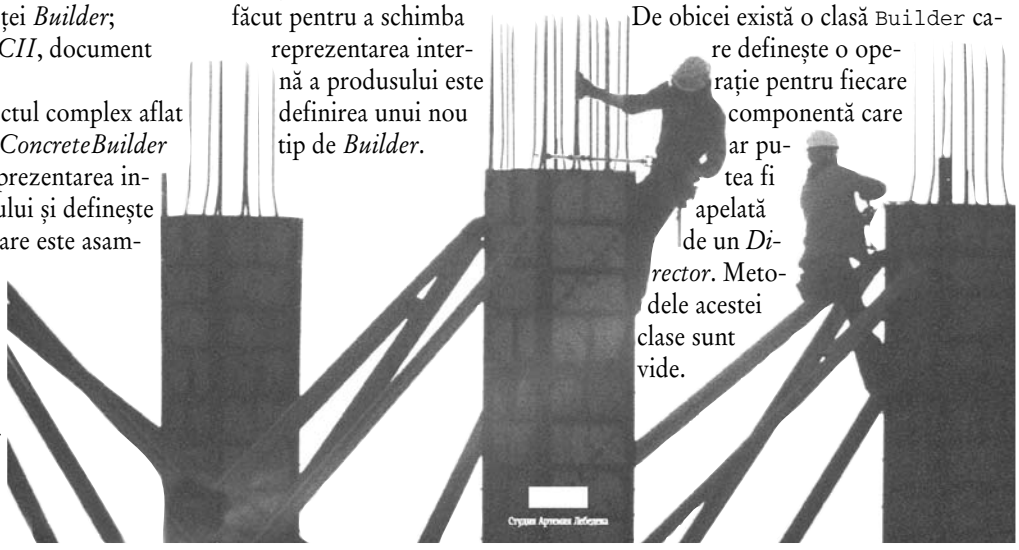
Consecințe

Principalele consecințe sunt următoarele:

- Permite modificarea reprezentării interne a produsului. Obiectul *Builder* furnizează o interfață abstractă pentru construirea produsului. Interfața permite *Builder*-ului să ascundă reprezentarea și structura internă a produsului. De asemenea, modul în care este asamblat produsul este ascuns. Deoarece produsul este construit printr-o interfață abstractă, tot ceea ce trebuie făcut pentru a schimba reprezentarea internă a produsului este definirea unui nou tip de *Builder*.
- Izolează codul pentru construcție și reprezentare. Modelul *Builder* îmbunătățește modularizarea prin încapsularea modului în care un obiect complex este construit și reprezentat. *Clienții* nu au nevoie să cunoască detalii despre clasele care definesc structura internă a produsului; astfel de clase nu apar în interfața *Builder*-ului. Fiecare *ConcreteBuilder* conține tot codul necesar creării și asamblării unui tip de produs. Codul este scris o singură dată, după care diferiți *Directori* pot să-l reutilizeze pentru a construi *Produse* din alte surse. Pentru exemplul anterior, am putea defini un cititor dintr-un format diferit de *RTF*, de exemplu *HTML*, și să utilizeze clasele `ASCIIConverter` sau `PDFConverter` pentru a extrage textul sau a crea documente *PDF*.
- Asigură un control fin asupra procesului de construcție. Spre deosebire de modelele care construiesc produsele într-o singură etapă, modelul *Builder* construiește produsul pas cu pas, sub controlul *Directorului*. Doar când produsul este terminat *Directorul* îl va obține de la *Builder*. Ca urmare, interfața *Builder* reflectă procesul de construcție a *Produsului* mai mult decât alte modele. Aceasta conferă mai mult control asupra procesului de construcție și ca urmare și asupra structurii interne a obiectului rezultat.

Implementare

De obicei există o clasă `Builder` care definește o operație pentru fiecare componentă care ar putea fi apelată de un *Director*. Metodele acestei clase sunt vide.



Clasele care sunt derivate din ea vor implementa doar operațiile pentru componentele de care sunt interesate.

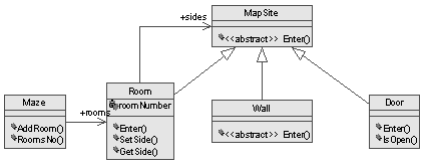
Alte probleme de implementare care se pot considera sunt:

- *Construcția și interfața pentru construirea Produsului*. Clasa `Builder` construiește obiectele pas cu pas. Ca urmare, interfața clasei `Builder` trebuie să fie suficient de generală pentru a permite construcția de produse pentru toate felurile de *Builder*-e. Lucrurile sunt simple atunci când construcția obiectului constă doar din adăugarea de elemente la sfârșitul unei colecții. Mai complicată este situația în cazul în care este necesar accesul la părți ale produsului, de exemplu când se construiesc arbori pornind de la bază (*bottom-up*). În acest caz, *Builder*-ul va returna noduri fii *Directorului*, care la rândul său le va transmite înapoi *Builder*-ului pentru a construi nodurile părinte.
- *Clasă abstractă pentru Produse*. În general, *Produsele* create de către *Builder*-e diferă suficient de mult în reprezentarea lor încât e prea puțin de câștigat prin generalizarea *Produselor*. Deoarece de obicei clientul este cel care configurează *Directorul* cu obiectul *Builder*, clientul este în măsură să recupereze produsul *Builder*-ului conform cu interfața acestuia.
- *Metode vide în implementarea implicită pentru Builder*. Metodele de creare definite de clasa *Builder* sunt în mod intenționat declarate virtuale, nu virtuale pure. În acest fel, subclasele trebuie să implementeze doar operațiile de care sunt interesate.

Exemplu

Pentru exemplificare vom implementa o funcție `CreateMaze` pentru a construi un labirint format din încăperi care va fi folosit pentru un joc. Fiecare încăpere conține detalii despre vecinii săi; vecinii pot fi o altă încăpere, un perete sau o ușă către o altă încăpere.

Clasele `Room`, `Door` și `Wall` definesc componentele labirintului. Relațiile între aceste clase sunt prezentate în următoarea diagramă.



Fiecare încăpere are patru vecini. Pentru a specifica direcțiile în care sunt plasați vecinii, vom folosi o enumerare:

```
enum Direction { North,
                 South, East, West };
```

Clasa `MapSite` este clasa abstractă comună tuturor componentelor labirintului. Pentru a simplifica exemplul, `MapSite` definește o singură metodă: `Enter`. Semnificația ei depinde de locul în care se intră. Dacă se intră într-o cameră, atunci locația jucătorului se schimbă. Dacă jucătorul intră într-o ușă, unul din următoarele două evenimente poate avea loc: dacă ușa este deschisă, atunci el va intra în camera următoare; dacă ușa este închisă, atunci jucătorul va fi penalizat.

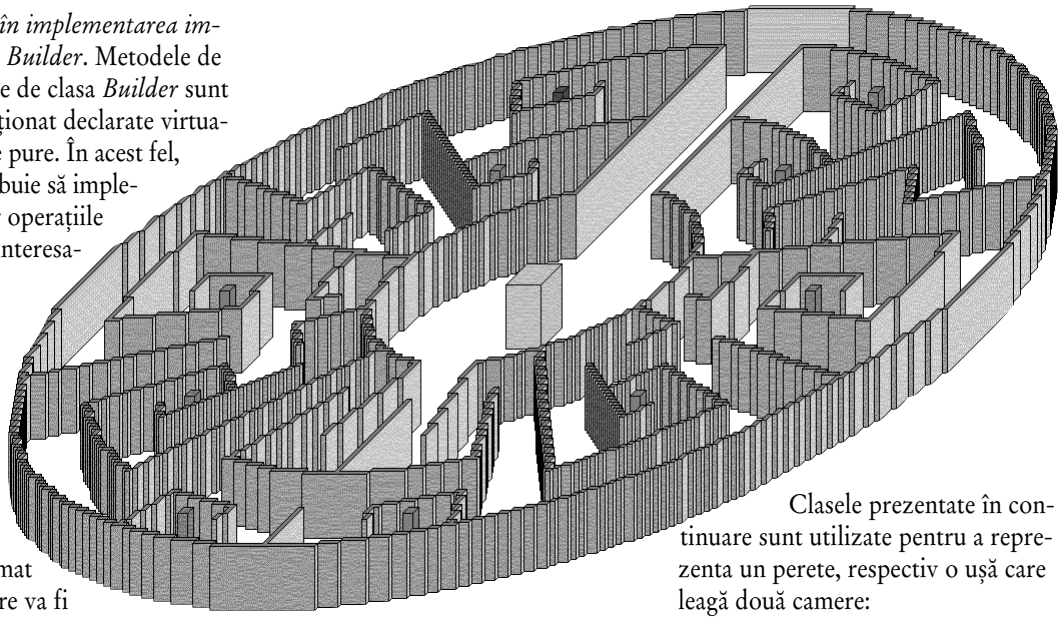
```
class MapSite {
public:
  virtual void Enter() = 0;
};
```

Metoda `Enter` furnizează o bază simplă pentru operațiile complexe ale jocului. De exemplu, dacă jucătorul se află într-o încăpere și decide să se deplaseze spre est, jocul poate să determine ce obiect `MapSite` se află în imediata vecinătate spre est și să apeleze metoda `Enter` a acelui obiect. Această operație va decide dacă locația jucatorului se schimbă sau el este penalizat.

`Room` este subclasa concretă a clasei `MapSite` care definește relația dintre componentele aflate în labirint. Ea menține referințe către alte obiecte `MapSite` și un număr care o va identifica în labirint.

```
class Room : public MapSite {
public:
  Room(int roomNo);

  MapSite* GetSide(Direction)
                          const;
  void SetSide(Direction,
                    MapSite*);
  virtual void Enter();
private:
  MapSite* sides[4];
  int roomNumber;
};
```



Clasele prezentate în continuare sunt utilizate pentru a reprezenta un perete, respectiv o ușă care leagă două camere:

```
class Wall : public MapSite {
public:
  Wall();

  virtual void Enter();
};

class Door : public MapSite {
public:
  Door(Room* = 0, Room* = 0);
  virtual void Enter();
  Room* OtherSideFrom(Room*);

private:
  Room* room1;
  Room* room2;
  bool isOpen;
};
```

Mai multe informații despre labirint sunt păstrate în clasa `Maze`. Aceasta va conține și o colecție de camere. `Maze` poate să găsească o încăpere pe baza numărului ei de ordine folosind metoda `RoomNo`.

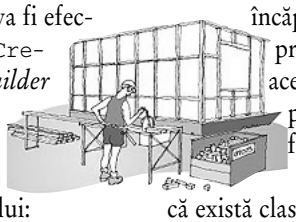
```
class Maze {
public:
  Maze();
  void AddRoom(Room*);
  Room* RoomNo(int) const;

private:
  // ...
};
```

O implementare a funcției `CreateMaze` care ignoră modelele de design ar putea fi următoarea:

```
Maze* CreateMaze () {
  Maze* aMaze = new Maze;
  Room* r1 = new Room(1);
  Room* r2 = new Room(2);
  Door* theDoor =
            new Door(r1, r2);
  aMaze->AddRoom(r1);
  aMaze->AddRoom(r2);
  r1->SetSide(North,
                  new Wall);
  r1->SetSide(East, theDoor);
  r1->SetSide(South,
                  new Wall);
  r1->SetSide(West,
                  new Wall);
  r2->SetSide(North,
                  new Wall);
  r2->SetSide(East,
                  new Wall);
  r2->SetSide(South,
                  new Wall);
  r2->SetSide(West, theDoor);
  return aMaze;
}
```

Revenind la modelul în discuție, construcția labirintului va fi efectuată de către o funcție `CreateMaze` folosind un *Builder* de tip `MazeBuilder`.



Clasa `MazeBuilder` definește interfața pentru construcția labirintului:

```
class MazeBuilder {
public:
  virtual void BuildMaze(){ }
  virtual void BuildRoom(
                  int room){ }
  virtual void BuildDoor(
                 int roomFrom,
                int roomTo){ }
  virtual Maze* GetMaze(){
    return 0;
  }

protected:
  MazeBuilder();
};
```

Această interfață poate crea trei elemente: labirintul (`Maze`), camere cu un anumit număr și uși între încăperi identificate prin numere. Metoda `GetMaze` furnizează labirintul construit.

Subclasele clasei `MazeBuilder` vor implementa această metodă pentru a furniza obiectul creat de ele.

Toate metodele de construcție ale clasei `MazeBuilder` sunt vide. Ele nu sunt declarate virtuale pure pentru a permite claselor derivate să implementeze doar metodele care sunt interesante pentru ele.

Folosind interfața `MazeBuilder`, funcția `CreateMaze` ar putea fi următoarea:

```
Maze* CreateMaze
       (MazeBuilder& builder){
  builder.BuildMaze();
  builder.BuildRoom(1);
  builder.BuildRoom(2);
  builder.BuildDoor(1, 2);

  return builder.GetMaze();
}
```

După cum se poate observa, *Builder*-ul ascunde reprezentarea internă a labirintului (clasele care definesc încăperile, pereții și ușile) precum și modul în care aceste părți sunt asamblate pentru a crea labirintul final.

Cineva ar putea ghici că există clase pentru încăperi și uși, dar nu există indicii despre pereți.

Aceasta ușurează modificarea modului în care labirintul este reprezentat, deoarece nici un client al clasei `MazeBuilder` nu trebuie modificat.

Clasa `StandardMazeBuilder` reprezintă o implementare care construiește labirinturi simple.

Ea păstrează labirintul care se construiește în variabila `currentMaze`.

```
class StandardMazeBuilder :
         public MazeBuilder {
public:
  StandardMazeBuilder();

  virtual void BuildMaze();
  virtual void
              BuildRoom(int);
  virtual void BuildDoor(int,
                         int);
  virtual Maze* GetMaze();

private:
  Direction CommonWall(Room*,
                       Room*);
  Maze* currentMaze;
};
```

Singura operație realizată de constructor este inițializarea variabilei membre:

```
StandardMazeBuilder::
      StandardMazeBuilder (){
  currentMaze = 0;
}
```

Metoda `BuildMaze` inițializează un obiect `Maze` pe care alte metode îl vor asambla sau îl vor furniza clienților.

```
void StandardMazeBuilder::
                BuildMaze () {
  currentMaze = new Maze();
}
```

Metoda `GetMaze` furnizează clienților interesați labirintul creat.

```
Maze* StandardMazeBuilder::
               GetMaze () {
  return currentMaze;
}
```

Metoda `BuildRoom` creează o încăpere și pereții din jurul ei:

```
void StandardMazeBuilder::
          BuildRoom (int n) {
  if (!currentMaze
                ->RoomNo(n)){
    Room* room = new Room(n);
    currentMaze
              ->AddRoom(room);
    room->SetSide(North, new
                    Wall());
    room->SetSide(South, new
                    Wall());
    room->SetSide(East, new
                    Wall());
    room->SetSide(West, new
                    Wall());
  }
}
```

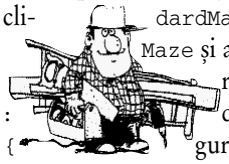
Pentru a construi o ușă între două încăperi, `StandardMazeBuilder` caută ambele încăperi și peretele dintre ele:

```
void StandardMazeBuilder::
   BuildDoor(int n1, int n2){
  Room* r1 =
     currentMaze->RoomNo(n1);
  Room* r2 =
     currentMaze->RoomNo(n2);
  Door* d = new Door(r1, r2);
  r1->SetSide(
       CommonWall(r1,r2), d);
  r2->SetSide(
       CommonWall(r2,r1), d);
}
```

Secvența completă pentru crearea labirintului este:

```
Maze* maze;
StandardMazeBuilder builder;
CreateMaze(builder);
maze = builder.GetMaze();
```

O altă abordare ar fi constat în plasarea întregului cod din `StandardMazeBuilder` în clasa `Maze` și acordarea labirinturilor a "permisiunii" de a se se construi singure.



Totuși, clasa `Maze`, fiind mai scurtă, este mai ușor de înțeles și modificat, iar clasa `StandardMazeBuilder` este ușor de separat de clasa `Maze`.

Mai important, separarea celor două clase ne permite să creăm o varietate de clase `MazeBuilder`, fiecare folosind clase diferite pentru pereți, încăperi și uși.

Mai interesantă este clasa `CountingMazeBuilder`. Această clasă nu construiește un labirint, ci doar numără cantitățile folosite din diferite componente (camera și uși) pentru a crea un labirint.

```
class CountingMazeBuilder :
          public MazeBuilder {
public:
  CountingMazeBuilder();

  virtual void BuildMaze();
  virtual void
               BuildRoom(int);
  virtual void BuildDoor(int,
                         int);

  void GetCounts(int&, int&)
                         const;

private:
  int doors;
  int rooms;
};
```

Constructorul inițializează cele două contoare, iar metodele suprâncărcate le incrementează corespunzător (pentru momentul în care elementele sunt utilizate):

```
CountingMazeBuilder::
      CountingMazeBuilder () {
  rooms = 0;
  doors = 0;
}

void CountingMazeBuilder::
            BuildRoom (int) {
  rooms++;
}

void CountingMazeBuilder::
       BuildDoor (int, int) {
  doors++;
}

void CountingMazeBuilder::
   GetCounts (int& r, int& d)
                       const {
  r = rooms;
  d = doors;
}
```

Iată cum ar putea fi folosită această clasă:

```
int rooms, doors;
CountingMazeBuilder builder;
CreateMaze(builder);
builder.GetCounts(rooms,
                       doors);
cout << "The maze has "
         << rooms
         << " rooms and "
         << doors
         << " doors"
         << endl;
```

Va urma...

Sperăm că prin acest model și exemplificarea lui v-am stârnit interesul pentru design în general și modele de design în particular. Vom continua în numerele următoare prezentarea celor mai cunoscute modele de proiectare.