Modele de PROIECTARE

Tudor Orha

În primul articol din acest serial am descris conceptul de model de proiectare, o clasificare a acestor modele și am început prezentarea modelelor creaționale cu modelul Builder. În acest al doilea episod al seriei vom continua prezentarea modelelor creaționale. Vor fi descrise alte două modele și anume: Singleton și Abstract Factory.

Singleton

Intenție

Obiectivul acestui model de proiectare este de a asigura faptul că o clasă are o singură instanță și să furnizeze un punct global de acces la această instanță.

Motivație

Este important ca unele clase să aibă exact o instanță. De exemplu, ar trebui să existe un singur sistem de fișiere și un singur gestionar de ferestre. Un alt exemplu este un filtru digital care trebuie să aibă un singur convertor analog/digital.

Cum ne putem asigura că o clasă are o singură instanță și că această instanță este ușor accesibilă? O variabilă globală face obiectul ușor accesibil, dar nu ne impiedică să instanțiem și alte obiecte.

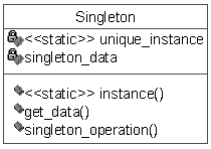
O soluție mai bună impune clasei să mențină unica sa instanță. Clasa poate asigura faptul că alte instanțe nu pot fi create (prin interceptarea cererilor de creare a obiectelor) și poate, de asemenea, să ofere o modalitate de acces la instanță. Acesta este modelul *Singleton*.

Aplicabilitate

Modelul *Singleton* este uilizat în următoarele situații:

- trebuie să existe o singură instanță dintr-o anumită clasă, iar aceasta trebuie să fie accesibilă printr-o modalitate bine-cunoscută;
- o singură instanță trebuie să fie extensibilă prin moștenire, iar clienții trebuie să poată să folosească instanța extinsă fără modificarea codului acestora din urmă.

Structura



Participanți

- *Singleton*:
  - definește o operație `instance` care permite clienților să acceseze unica instanță a clasei; această operație este o operație a clasei (statică în C++); ea poate fi responsabilă și pentru crearea acestei instanțe unice.

Colaborări

Clienții accesează instanța clasei `Singleton` numai prin intermediul operației `instance`.

Consecințe

Modelul *Singleton* prezintă mai multe beneficii:

- *controlarea accesului la instanța unică*; deoarece clasa `Singleton` încapsulează singura sa instanță, ea poate controla strict când și cum poate să fie accesată această instanță;
- *spațiu de nume (namespace) redus*; modelul *Singleton* reprezintă o îmbunătățire față de variabilele globale; el evită "poluarea" spațiului de nume cu variabile globale care conțin instanțe unice;
- *permiterea rafinării operațiilor și a reprezentării*; clasa `Singleton` poate fi moștenită și este ușor ca aplicația să fie configurată folosindu-se o instanță a unei clase derivate; este posibil ca această configurare să aibă loc în momentul execuției (*runtime*);
- *permiterea unui număr variabil de instanțe*; modelul ușurează schimbarea deciziei și permiterea a mai mult de o instanță a clasei `Singleton`; mai mult, folosind această abordare se poate controla numărul de instanțe pe care aplicația le folosește; numai operația care furnizează accesul la instanța clasei `Singleton` trebuie să fie schimbată;
- *flexibilitate mărită*; există și alte modalități de a implementa funcționalitatea modelului *Singleton* printre care folosirea operațiilor de clasă (metode statice în C++); această tehnică duce la o dificultate mai mare în schimbarea numărului de instanțe permise; mai mult, metodele statice nu pot fi virtuale, deci polimorfismul nu poate fi utilizat în cazul unei astfel de abordari.

Implementare

Câteva dintre problemele care pot apărea la implementarea modelului sunt următoarele:
- asigurarea unei instanțe unice;
- subclasarea clasei `Singleton`.

În cadrul modelului *Singleton* singura instanță a clasei este o instanță obișnuită, dar clasa este implementată în așa fel încât doar o singură instanță să poată fi creată.

O modalitate uzuală pentru a obține acest efect este ascunderea operației care creează instanța în spatele unei operații de clasă (statică) care să garanteze că numai o instanță va fi creată.

Această operație are acces la variabila care conține instanța unică și ne asigură că variabila este inițializată cu instanța unică înainte de a returna valoarea ei. Această abordare ne asigură că un *singleton* este inițializat înainte de a fi folosit.

Operația clasei poate fi definită ca o funcție membră statică `instance` pentru clasa `Singleton`. Această clasă va defini, de asemenea, și o variabilă membră statică `unique_instance` care va conține o referință către unica instanță a clasei.

Clasa `Singleton` poate fi declarată astfel:

```
class Singleton {
public:
  static Singleton*
                    instance();

protected:
  Singleton();

private:
  static Singleton*
              unique_instance;
};
```

Implementarea corespunzătoare este:

```
Singleton* Singleton::
          unique_instance = 0;

Singleton* Singleton::
                    instance(){
  if (unique_instance == 0){
    unique_instnace = new
                    Singleton();
  }

  return unique_instance;
}
```

Clienții accesează clasa `Singleton` doar prin intermediul metodei `instance`. Variabila `unique_instance` este inițializată cu `0`, iar metoda `instance` furnizează valoarea ei, inițializând-o în cazul în care valoarea este `0`.

Metoda `instance` folosește inițializarea târzie (*lazy initialization*): valoarea pe care o furnizează este creată și păstrată doar în momentul primului apel.

Se poate observa cum este protejat constructorul prin declararea lui ca **`protected`**. În felul acesta, dacă un client va încerca să instanțieze direct clasa `Singleton`, se va obține o eroare în momentul compilării codului.

Aceasta ne asigură că doar o singură instanță este creată. Mai mult, datorită faptului că variabila `unique_instance` este o referință spre un obiect `Singleton`, metoda `instance` poate accesa o referință la o instanță a unei subclase a clasei `Singleton`.

Mai trebuie menționat faptul că nu este suficient să definim instanța unică ca fiind un obiect global sau static și să ne bazam pe inițializarea automată.

Există trei motive pentru care nu este convenabil să folosim obiecte globale sau statice:
- nu se poate garanta că va fi creată doar o singură instanță;
- s-ar putea ca în momentul inițializării variabilelor statice să nu fie disponibile suficiente informații pentru crearea instanței; există posibilitatea ca unele informații să fie calculate mai târziu, în decursul execuției programului.
- limbajul C++ nu definește ordinea în care sunt apelați constructorii pentru variabilele globale; aceasta înseamnă că între diferite instanțe ale diverselor clase *Singleton* nu pot exista dependințe; dacă acestea există, atunci problemele sunt inevitabile.

O altă problemă cauzată de abordarea bazată pe variabilele globale sau statice este dată de faptul că se forțează ca toate aceste variabile (pentru toate clasele care implementează modelul *Singleton*) să fie instanțiate. Folosirea unei metode statice înlătură toate aceste probleme.

În cazul subclasării clasei *Singleton*, principala problema nu o constituie definirea unei clase derivate, ci instalarea instanței unice a acestei subclase astfel încât clienții să o poată utiliza.

În esență, variabila care referă instanța unică trebuie inițializată cu o instanță a subclasei.

Metoda cea mai simplă este de a determina care instanță să fie folosită în operația `instance` a clasei `Singleton`.

În secțiunea dedicată exemplelor vom arăta cum se realizează acest lucru folosind variabilele de mediu (*environment variables*).

O altă posibilitate pentru alegerea subclasei *singleton* este mutarea implementării metodei `instance` din clasa părinte în subclasă.

Aceasta îi permite programatorului C++ să decidă clasa unui *singleton* în momentul linkeditării (prin legarea unui fișier obiect conținând o implementare diferită), iar această decizie rămâne ascunsă clienților clasei.

Soluția precedentă impune alegerea clasei în momentul linkeditării, ceea ce face dificilă alegerea clasei în momentul execuției.

Utilizarea instrucțiunilor condiționale pentru a alege clasa este mai flexibilă, dar impune doar anumite variante de alegere.

În concluzie, nici una din aceste posibilități nu este suficient de flexibilă.

O metodă care oferă un grad acceptabil de flexibilitate constă în folosirea unui registru de instanțe *singleton*. În loc să definim în metoda `instance` mulțimea claselor *singleton*, chiar clasa `Singleton` poate să înregistreze (folosind un nume unic) instanța unică într-un registru prestabilit.

Registrul creează o asociere între denumiri și instanțe *singleton*. În momentul în care metoda `instance` are nevoie de o instanță, ea va consulta registrul folosind numele pentru a identifica instanța.

Registrul va căuta instanța corespunzătoare și o va furniza (dacă aceasta există). Astfel, metoda `instance` nu trebuie să cunoască toate posibilele clase sau instanțe *singleton*. Ea are nevoie doar de o interfață comună pentru toate subclasele *singleton* care să includă operațiile pentru registru.

Prezentăm în continuare modul în care este declarată clasa `Singleton` dacă este utilizată abordarea descrisă anterior:

```
class Singleton {
public:
  static void register(
                  char *name,
        Singleton *instance);
  static Singleton*
                  instance();

protected:
  static Singleton* lookup(
           const char* name);

private:
  static Singleton*
             unique_instance;
  static List* registry;
};
```

Metoda `register` va înregistra o instanță sub un anumit nume. Pentru a păstra lucrurile simple, vom folosi o listă *STL* de perechi `pair<char*, Singleton*>`.

Fiecare element al listei va asocia un anumit nume unei anumite instanțe.

Metoda `lookup` caută o instanță pe baza numelui asociat ei.

Vom presupune că o variabilă de mediu conține numele instanței care se dorește a fi folosită.

Implementarea metodei `instance` este prezentată în cele ce urmează:

```
Singleton* Singleton::
                  instance(){
  if (unique_instance == 0){
    const char*
            singleton_name =
          getenv("SINGLETON");
  unique_instance =
      lookup(singleton_name);
  }

  return unique_instance;
}
```

Subclasele clasei `Singleton` își pot înregistra instanțele în constructor.

De exemplu, constructorul unei clase `MySingleton` ar putea arăta astfel:

```
MySingleton::MySingleton(){
  // ...
  Singleton::register(this);
}
```

Bineînțeles, constructorul clasei `MySingleton` nu va fi apelat decât în momentul construirii unei instanțe, adică exact ceea ce acest model încearcă să rezolve! Această problemă se poate ocoli în C++ prin definirea unei instanțe statice.

De exemplu, putem include declarația

```
static MySingleton
                  theSingleton;
```

în fișierul care conține implementarea clasei `MySingleton`.

Astfel, metoda `instance` a clasei `Singleton` nu mai are rolul de a crea instanța unică, ci rolul de a alege care instanță să fie folosită în sistem.

Soluția obiectelor statice are în continuare dezavantajul instanțierii tuturor subclaselor *singleton* pentru că altfel acestea nu vor fi înregistrate.

Exemple

Presupunem că dorim să creăm o clasă `MazeFactory` (vezi și modelul *Abstract Factory* care va fi prezentat tot în cadrul acestui episod) care să creeze părțile componente ale unui labirint.

Subclasele ei pot reimplementa unele dintre operațiile definite de `MazeFactory` pentru a schimba tipul unora dintre elementele labirintului.

De exemplu, `EnchantedMazeFactory` ar putea crea instanțe `EnchantedRoom` în locul simplelor instanțe `Room` create de `MazeFactory`.

Ceea ce este important în cazul de față este faptul că aplicația are nevoie doar de o singură instanță a clasei `MazeFactory` și că această instanță trebuie să fie disponibilă codului care creează labirintul.

Implementând clasa `MazeFactory` folosind modelul *singleton*, ne vom asigura de accesul global la o instanță unică a acestei clase.

Pentru ca `MazeFactory` să respecte modelul *singleton*, trebuie să modificăm ușor clasa:

```
class MazeFactory {
public:
  static MazeFactory*
                  instance();
  // definirea celorlalte metode ale
  // interfeței

protected:
  MazeFactory();
  // constructorul nu trebuie să fie
  // accesibil clienților
```

```
private:
  static MazeFactory*
              unique_instance;
};
```

Implementarea se modifică și ea prin adăugarea secvenței următoare:

```
MazeFactory* MazeFactory::
         unique_instance = 0;

MazeFactory* MazeFactory::
                  instance(){
  if (unique_instance == 0){
    unique_instance = new
               MazeFactory();
  }

  return unique_instance;
}
```

Pentru simplitate, am considerat ca nu vor exista subclase ale clasei `MazeFactory`.

Abstract Factory

Intenția

Acest model are ca scop furnizarea unei interfețe pentru crearea unor familii de obiecte dependente sau între care există diferite alte tipuri de relații.

Motivație

Putem considera un set de clase utilizate pentru a interacționa cu utilizatorul care să suporte multiple standarde pentru aspect (*look and feel*), cum ar fi *Motif* sau *Presentation Manager*.

Diferitele standarde definesc aspecte și comportamente pentru elemente de interfață cu utilizatorul: ferestre, butoane, bare de rulare etc.

Pentru a fi compatibilă cu diferitele standarde, o aplicație trebuie să nu folosească direct elementele de interfață ale unui anumit standard.

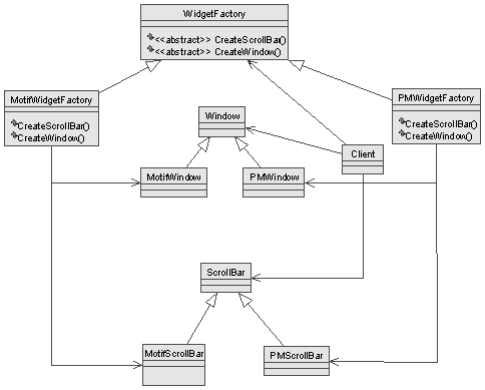
Instanțierea claselor specifice unui anumit standard în întregul cod al aplicației va face dificilă schimbarea standardului mai târziu.

Această problemă se poate rezolva prin definirea unei clase abstracte `WidgetFactory` care să declare o interfață pentru crearea fiecărui tip de element de interfață.

De asemenea există o clasă abstractă pentru fiecare tip de element de interfață și clase concrete care implementează elementele de interfață pentru fiecare standard în parte.

În interfața `WidgetFactory` se definește câte o operație pentru fiecare tip de element de interfață, care furnizează noi instanțe de elemente de interfață.

Clienții vor apela aceste metode pentru a obține elemente de interfață, fără a cunoaște clasele concrete care sunt folosite. Așadar, clienții rămân independenți de standardul de aspect care este folosit.



Pentru fiecare standard de aspect există o subclasă concretă a clasei `WidgetFactory`.

Fiecare subclasă implementează operațiile care creează elementele de interfață potrivite pentru respectivul standard.

De exemplu, operația `CreateScrollBar` a clasei `MotifWidgetFactory` ar putea instanția și furniza o bară de rulare specifică aspectului *Motif*, în timp ce aceeași operație a clasei `PMWidgetFactory` ar instanția și furniza o bară de rulare specifică aspectului *Presentation Manager*.

Clienții creează elemente de interfață numai prin intermediul interfeței `WidgetFactory` și nu dețin absolut nici o informație despre clasele care implementează elementele de interfață specifice unui anumit standard.

O clasă `WidgetFactory` asigură și dependența dintre clasele concrete care implementează elementele de interfață.

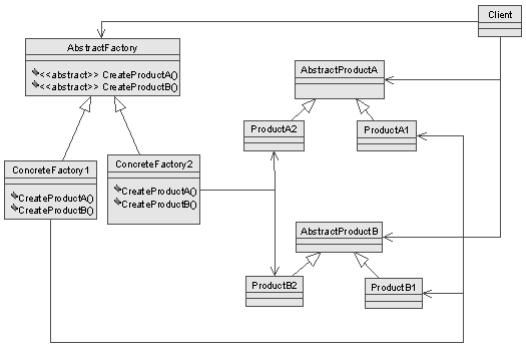
De exemplu, o bară de stare cu aspectul *Motif* ar trebui să fie utilizat doar împreună cu un buton *Motif* sau cu un editor de texte *Motif*, iar această cerință este asigurată automat ca o consecință a folosirii unei clase `MotifWidgetFactory`.

Aplicabilitate

Câteva dintre situațiile în care se poate folosi modelul *Abstract Factory* sunt următoarele:

- un sistem trebuie să fie independent de modul în care produsele sale sunt create, compuse și reprezentate;
- un sistem trebuie să fie configurat cu una din mai multe familii de produse;
- o familie de produse trebuie să fie folosite împreună și este necesară asigurarea acestei cerințe;
- se dorește furnizarea unei librării de clase, dar în același timp se dorește dezvăluirea doar a interfețelor acestor clase, nu și implementarea propriu-zisă.

Structura



Participanți

- *AbstractFactory*:
  - declară o interfață pentru operații care creează produse (obiecte) abstracte;
- *ConcreteFactory*:
  - implementează operațiile din *AbstractFactory*; aceste operații vor furniza produse (obiecte) concrete;

- *AbstractProduct*:
  - ♦ declară o interfață pentru un tip de produs (obiect);
- *ConcreteProduct*:
  - ♦ definește un produs (obiect) concret care să fie creat de către clasa *ConcreteFactory* corespunzătoare;
  - ♦ implementează interfața *AbstractProduct*
- Client:
  - ♦ folosește (numai) interfețele *AbstractFactory* și *AbstractProduct*.

Colaborări

De obicei este creată o singură instanță a unei clase *ConcreteFactory*.

Această "fabrică" va crea produse având o anumită implementare. Pentru a crea obiecte specifice altor implementări, clienții trebuie să folosească "fabrici" diferite.

Clasa *AbstractFactory* transferă responsabilitatea creării obiectelor subclaselor sale concrete.

Consecințe

Modelul *Abstract Factory* are următoarele avantaje și dezavantaje:

- *izolarea claselor concrete*; modelul *Abstract Factory* oferă un control sporit asupra claselor de obiecte create de aplicație; deoarece o "fabrică" are responsabilitatea creării obiectelor, ea izolează clienții de clasele care implementează o anumită funcționalitate; clienții manipulează instanțele doar prin intermediul interfeței abstracte asociate. Numele propriu-zise ale claselor concrete sunt izolate în implementarea clasei "fabrică" concrete, ele neapărând în codul client.

- *ușurință în schimbarea claselor concrete*; clasa unei "fabrici" concrete apare într-un singur loc în aplicație și anume la instanțierea ei; se pot folosi diferite configurații de produse (obiecte) schimbând doar clasa "fabrică" concretă; datorită faptului că o "fabrică" abstractă creează o întreagă familie de obiecte, întreaga familie se schimbă dacă este schimbată "fabrica"; în exemplul cu cele două standarde de interfață, putem schimba aspectul aplicației prin schimbarea clasei "fabrică" și recrearea interfeței aplicației.

- *consistența între obiecte*; când obiectele dintr-o familie sunt proiectate să funcționeze împreună, este important ca o aplicație să folosească obiecte dintr-o singură familie la un moment dat; folosind modelul *Abstract Factory* această cerință este foarte ușor de respectat.
- *dificultate la adăugarea de noi tipuri de produse (obiecte)*; extinderea "fabricilor" abstracte nu este o operație simplă datorită faptului că interfața *AbstractFactory* stabilește clar mulțimea de obiecte care pot fi create; adăugarea unui tip nou de obiect la interfață necesită extinderea acesteia, împreună cu toate subclasele ei; o posibilă soluție va fi prezentată în secțiunea dedicată implementării.

Implementare

În continuare vom prezenta câteva tehnici utile în implementarea modelului *Abstract Factory*:

- "fabrici" care folosesc modelul *Singleton*;
- crearea produselor;
- definirea unor "fabrici" extensibile.

O aplicație are nevoie în general de o singură instanță a unei "fabrici" concrete pentru o familie de obiecte. Ca urmare, ea este cel mai bine implementată ca un *singleton* (acest model de proiectare a fost prezentat în cadrul acestui episod).

Clasa `AbstractFactory` doar declară o interfață pentru crearea de obiecte. Este responsabilitatea subclaselor *ConcreteFactory* să le creeze efectiv. Cea mai obișnuită metodă este declararea unei operații pentru fiecare tip de produs (obiect). O "fabrică" concretă va specifica produsele sale prin redefinirea fiecărei operații.

Deși această implementare este simplă, ea necesita o nouă subclasă "fabrică" concretă pentru fiecare familie de obiecte, chiar dacă familiile diferă foarte puțin între ele.

În cazul în care există mai multe posibile familii de produse, clasa "fabrică" concretă poate fi implementată folosind modelul *Prototype*.

"Fabrica" este inițializată cu câte o instanță prototip pentru fiecare tip de produs (obiect) a familiei și va crea noi instanțe prin clonarea prototipului corespunzător. În acest fel se elimină necesitatea unei noi clase pentru fiecare familie de produse.

Folosind limbajul C++, vom prezenta o posibilă implementare a unei astfel de "fabrici". În această implementare, prototipurile vor fi păstrate într-un dicționar, asociate unui nume unic.

Adăugarea de prototipuri la "fabrică" se realizează cu ajutorul metodei `add_part`, iar obținerea unui produs (obiect) se

face prin apelul metodei `make_part`. Pentru simplitate, nu vom prezenta validările datelor de intrare.

```
class ConcreteFactory :
      public AbstractFactory {
map<string, AbstractProduct*>
                    dictionary;
public:
  void add_part(string name,
              AbstractProduct*
                    prototype){
    dictionary[name] =
          prototype->clone();
  }

  AbstractProduct*
      make_part(string name){
    return dictionary[name]
                    ->clone();
  }

};
```

Clasa `AbstractFactory` declară de obicei operații diferite pentru fiecare tip de produs pe care îl poate furniza. Tipurile produselor sunt precizate în semnăturile operațiilor. Adăugarea unui nou tip de produs necesită atât schimbarea clasei `AbstractFactory`, cât și a tuturor subclaselor sale.

O proiectare mai flexibilă, dar mai puțin sigură, este adăugarea unui parametru la operațiile care creează obiecte. Acest parametru specifică tipul obiectului care va fi creat. Ar putea fi un identificator de clasă, un întreg, o denumire sau orice altceva ce identifica tipul de produs. De fapt, folosind această abordare, `AbstractFactory` are nevoie de o singură metodă `make` cu un parametru care să indice tipul de obiect care să fie creat. Aceasta este și tehnica bazată pe prototipuri descrisă anterior.

Această metodă poate fi folosită în C++ doar când toate obiectele au aceeași clasă de bază (abstractă) sau când obiectele create pot fi convertite în mod sigur la tipul cerut de client.

Dar, chiar și dacă nici o conversie nu este necesară, o problemă inerentă tot rămâne: toate produsele furnizate clienților au aceeași interfață abstractă dată de tipul returnat.

Clientul nu va fi capabil să diferențieze sau să facă presupuneri sigure despre clasa unui produs.

Dacă clienții au nevoie să execute operații specifice subclaselor, acestea nu vor fi accesibile prin intermediul interfeței abstracte.

Deși clienții pot efectua o conversie spre subclase (folosind operatorul *dynamic_cast* în C++), aceasta nu este întotdeauna sigură sau fezabilă, deoarece conversia poate eșua. Acesta este compromisul clasic al unei interfețe foarte flexibile și extensibile.

Exemple

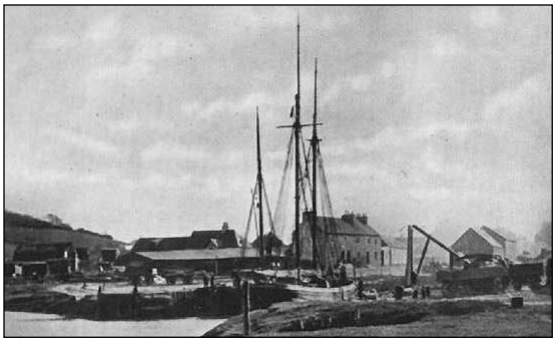
Vom aplica modelul *Abstract Factory* pentru a crea labirintul prezentat în cadrul primului episod al acestui serial.

Clasa `MazeFactory` poate crea componente ale labirintului. Va construi încăperi, pereți și uși. Ea ar putea fi folosită de către un program care citește planuri de labirinturi dintr-un fișier si construiește labirinturile corespunzătoare sau de un program care creează labirinturi aleatoare.

Programele care creează labirintul primesc un parametru `MazeFactory` care va permite programatorului să specifice clasele pentru încăperi, pereți și uși care vor fi construite.

```
class MazeFactory {
public:
  MazeFactory() {}

  virtual Maze* make_maze()
                          const{
    return new Maze();
  }

  virtual Room* make_room(
                int n) const{
    return new Room(n);
  }

  virtual Door* make_door(
          Room* ra, Room* rb)
                          const{
    return new Door(ra, rb);
  }

  virtual Wall* make_wall()
                          const{
    return new Wall();
  }

};
```

Vă reamintim că funcția `CreateMaze` construiește un mic labirint format din două încăperi cu o ușă între ele. Această funcție



conține numele claselor folosite, făcând dificilă crearea unor labirinturi folosind alte componente.

În continuare vom prezenta o variantă a funcției `CreateMaze` care remediaza acest neajuns prin intermediul unui parametru de tip `MazeFactory`.

```
Maze* CreateMaze(
        MazeFactory& factory){
  Maze* aMaze =
         factory.make_maze();

  Room* r1 =
        factory.make_room(1);
  Room* r2 =
        factory.make_room(2);

  Door* theDoor =
    factory.make_door(r1, r2);

  aMaze->AddRoom(r1);
  aMaze->AddRoom(r2);

  r1->SetSide(North,
        factory.make_wall());
  r1->SetSide(East, theDoor);
  r1->SetSide(South,
        factory.make_wall());
  r1->SetSide(West,
        factory.make_wall());

  r2->SetSide(North,
        factory.make_wall());
  r2->SetSide(East,
        factory.make_wall());
  r2->SetSide(South,
        factory.make_wall());
  r2->SetSide(West, theDoor);

  return aMaze;

}
```

Putem crea o clasa `EnchantedMazeFactory`, o "fabrică" pentru labirinturi magice, subclasând `MazeFactory`. `EnchantedMazeFactory` va redefini unele metode și va furniza subclase care vor înlocui `Room` și `Wall`.

```
class EnchantedMazeFactory :
          public MazeFactory {
public:
  EnchantedMazeFactory() {}

  virtual Room* make_room(
                  int n) const{
    return new EnchantedRoom(
             n, cast_spell());
  }

  virtual Door* make_door(
   Room* ra, Room* rb) const{
    return new
    DoorNeedingSpell(ra, rb);
  }
protected:
  Spell* cast_spell() const;
};
```

Pentru a crea un labirint care să folosească încăperi magice și uși care au nevoie de vrăji pentru a fi deschise, se va folosi nouă clasă "fabrică", astfel:

```
{
  ...

  EnchantedMazeFactory
                      factory;

  Maze* maze =
        CreateMaze(factory);

  ...
}
```

Se poate observa că `MazeFactory` este doar o colecție de metode "fabrică".

Acesta este modul obișnuit de implementare al modelului *Abstract Factory*.

De asemenea, se observă că `MazeFactory` nu este o clasă abstractă, jucând astfel atât rolul de *AbstractFactory*, cât și de *ConcreteFactory*.

Aceasta este o altă implementare obișnuită pentru utilizări simple ale modelului *Abstract Factory*.