# *Proiectarea de detaliu -2*

*Prof. univ. dr. ing. Florica Moldoveanu*

Curs *Ingineria programelor* – UPB, Automatică şi Calculatoare
2020-2021

# Șablonul Abstract Factory (1)

**Nume:** Abstract Factory; **tipul**: creațional

**Descrierea problemei:** Posibilitatea de a lucra cu un set de concepte (produse abstracte) pentru care sunt posibile mai multe implementări, fără a depinde de implementarile concrete.

De exemplu, toate platformele software implementează un acelaşi set de elemente de interfață utilizator: buton, checkbox, listbox, etc. Acestea reprezintă concepte (produse abstracte), căci funcționalitațile lor sunt aceleaşi, independente de modul de implementare.

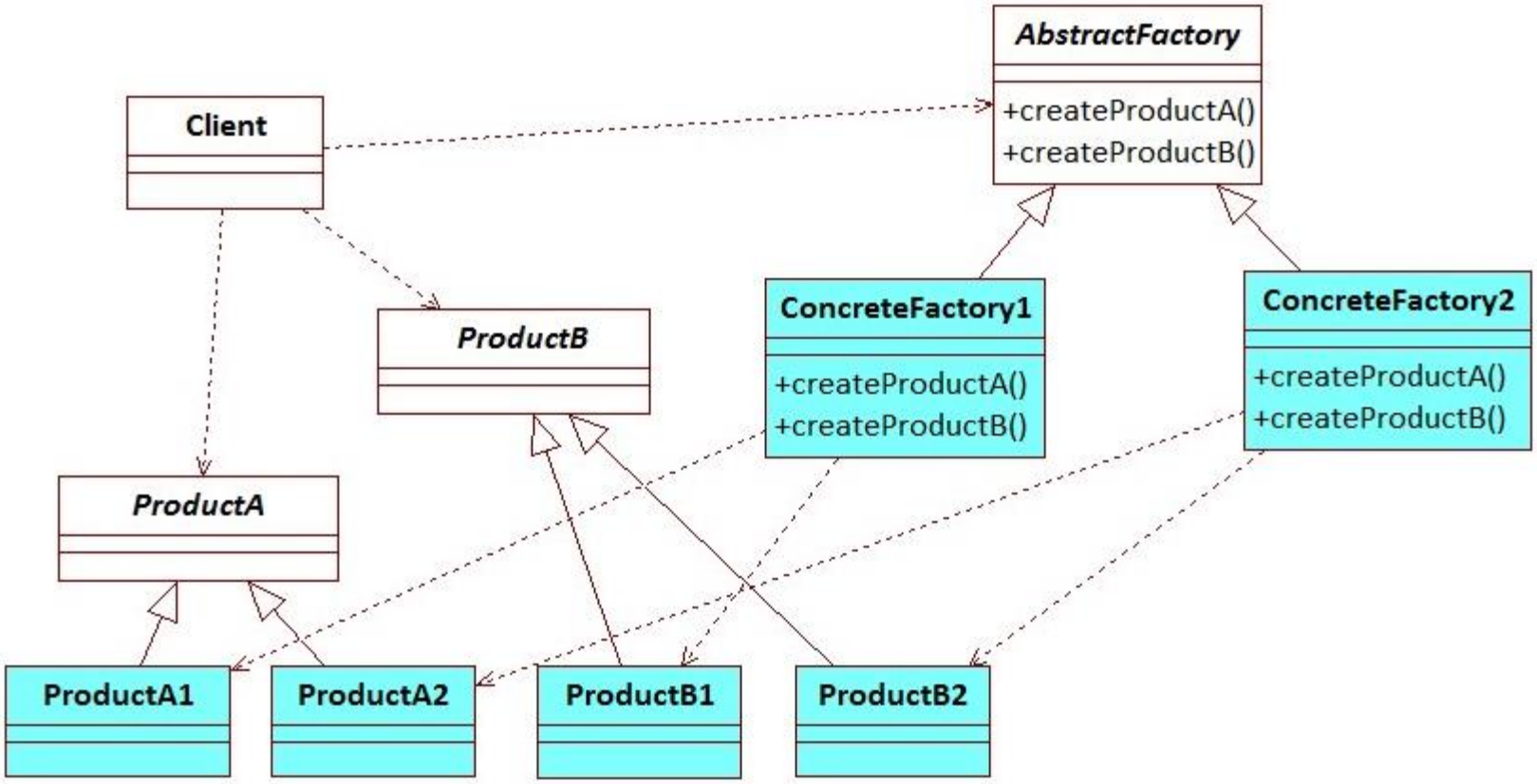
**Soluția**

Fiecare **produs abstract** dintr-o familie **este reprezentat printr-o interfață, *Product*.**

**O platformă abstractă** – *AbstractFactory***,** este reprezentată de un set de produse abstracte. Interfața *AbstractFactory* declară operațiile de creare a produselor abstracte individuale: CreateProductA(), CreateProductB(), etc.

**O platformă specifică** este reprezentată printr-o **clasă *ConcreteFactory*** şi un set de clase care implementează **produsele proprii platformei** (câte una pentru fiecare produs abstract).

# Șablonul Abstract Factory (2)




**Consecințe:** clientul este independent de clasele produselor concrete, independent de platforma.

- Este posibila înlocuirea familiilor de obiecte în timpul execuției.
- Adaugarea de produse noi este dificilă deoarece trebuie extinse toate clasele ConcreteFactory.

# Șablonul Abstract Factory (3)

**Exemplu:**

O aplicație pentru o casă inteligentă foloseste diferite tipuri de senzori (de temperatură, de fereastră deschisă/inchisă, bec aprins/stins, etc), identifică anumite conditii predefinite si trimite comenzi catre dispozitive de actionare (aer conditionat on/off, declanşarea unei alarme, etc).

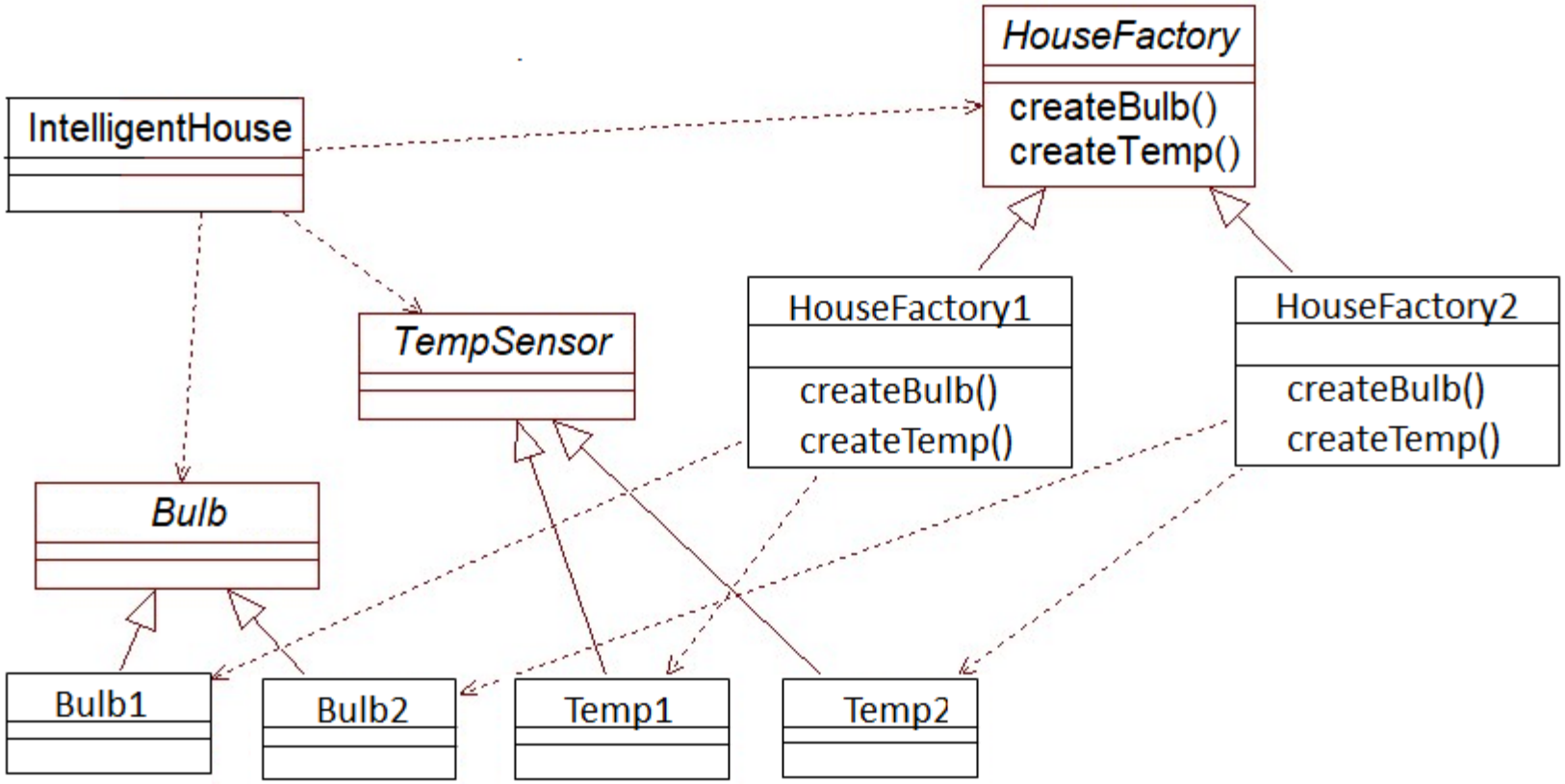
Aplicația nu trebuie să depindă de particularitătile de realizare a fiecărui dispozitiv utilizat, specifice fiecărui fabricant, de aceea lucrează cu dispozitive generice, produse abstracte (*Bulb*, *TempSensor*).

Fiecare fabricant este reprezentat printr-o clasă (HouseFactory1, HouseFactory2) care furnizează metodele de creare a dispozitivelor abstracte (CreateBulb, CreateTemp)**.**

Clasele Client accesează numai interfețele *HouseFactory, Bulb* şi *TempSensor*, fiind astfel protejate de particularitatile dispozitivelor produse de diverşi fabricanți.

# Șablonul Abstract Factory (4)

Diagrama de clase a aplicației **Casa inteligentă**



```
#include <iostream>
class Bulb //clasa abstracta
{  public:
   virtual float cost() = 0;
};

class HouseFactory// clasa abstracta
{ public:
  virtual Bulb * createBulb() = 0;
//virtual TempSensor * createTemp()=0;
};

class Bulb1 : public Bulb
{ public:
  Bulb1() {}
  float cost() { return 3.5; }
};

class HouseFactory1:public HouseFactory
{ public:
  HouseFactory1() {}
  Bulb * createBulb()
  { return new Bulb1(); }
 // TempSensor * createTemp(){…};
};
```

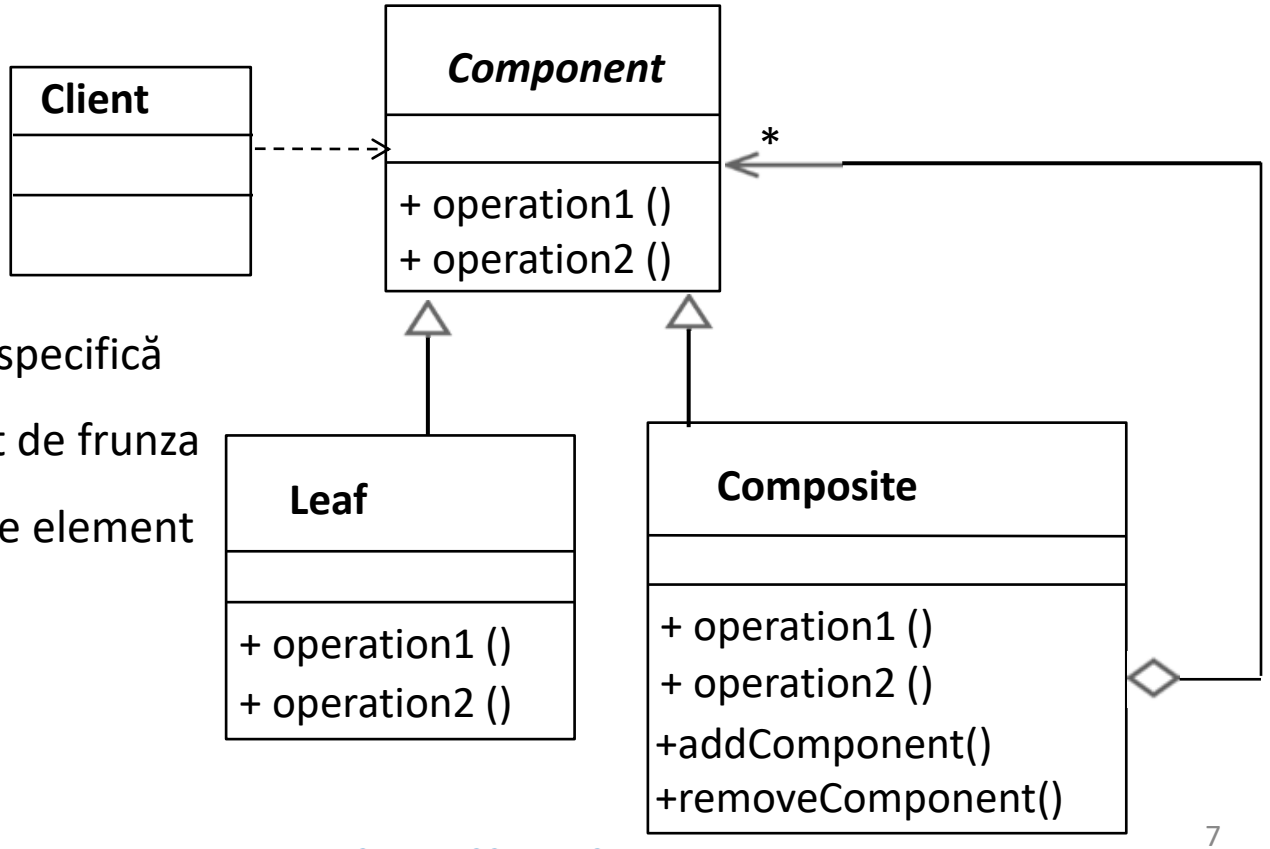
```
class IntelligentHouse
{ HouseFactory * hF;
  Bulb * aBulb; // TempSensor * aTemp;
  public:
  IntelligentHouse
  (HouseFactory *_hF){ hF = _hF;}

  void configurate()
  { aBulb = hF->createBulb();
   //aTemp = hF->createTemp();
  }

  float cost()
  { return aBulb->cost()/*+ aTemp->cost()*/;}
};

int main()
{HouseFactory * aFactory = new  HouseFactory1();
IntelligentHouse * anIntelligentHouse = new
IntelligentHouse(aFactory);
anIntelligentHouse->configurate();
std::cout << anIntelligentHouse->cost();
……
}
```

# Şablonul Composite (1)

**Reprezentarea ierarhiilor recursive**

**Nume:** Composite; **tipul**: structural

**Descrierea problemei:** reprezentarea unei ierarhii de înaltime şi lățime variabile astfel încât frunzele şi agregatele să poată fi tratate uniform, prin aceeasi interfață.




**Solutia:** Interfața *Component* specifică serviciile care sunt oferite atât de frunza cât şi de agregat pentru fiecare element componenta sa.

# Șablonul Composite (2)

- Agregatul implementează fiecare serviciu pentru fiecare componenta pe care o conține.
- Obiectele frunza (Leaf) implementează efectiv serviciile (Leaf.move(x,y)).

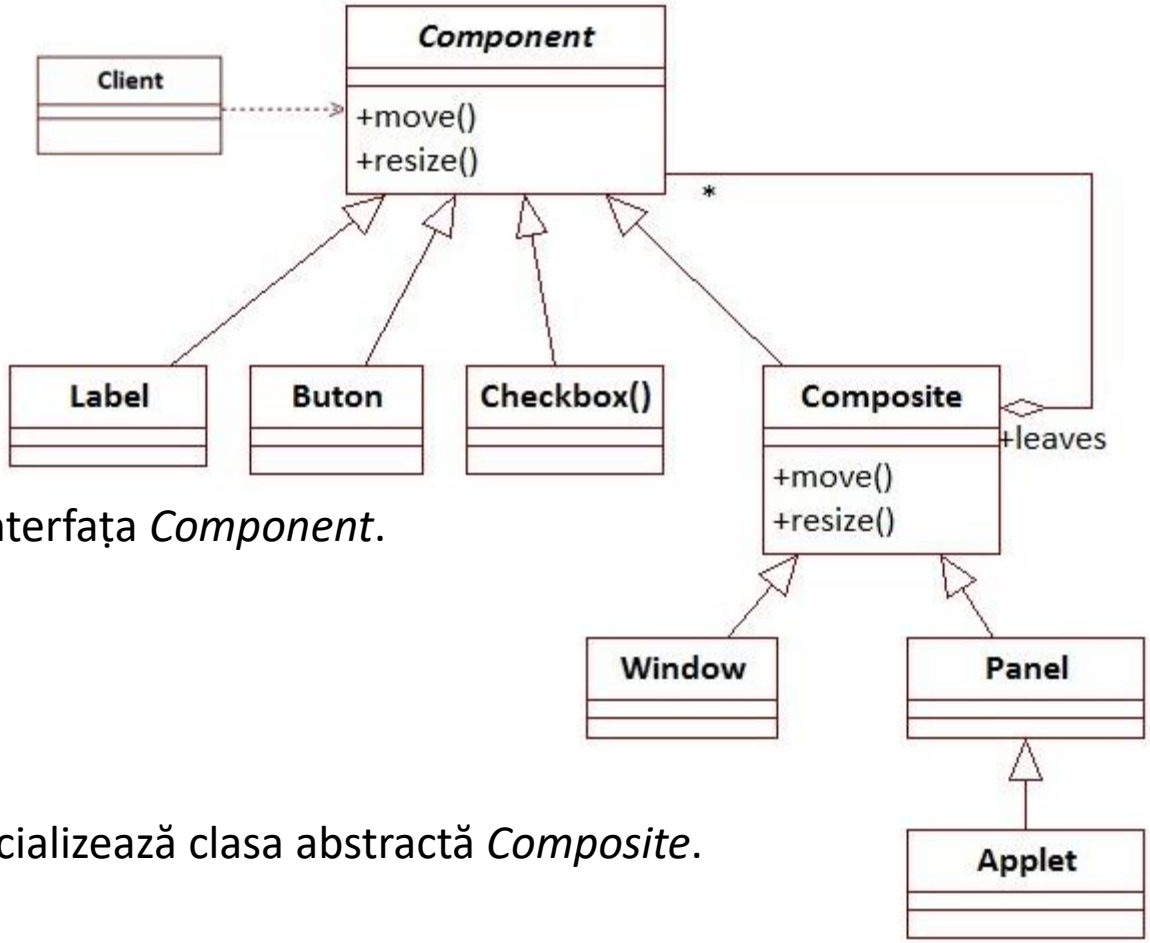
**Consecințe:**

- Clienții utilizează acelaşi cod pentru lucrul cu obiectele frunză şi cu obiectele agregat.
- Comportarea implementată în clasa Leaf poate fi modificată fără schimbarea ierarhiei.
- Pot fi adăugate noi clase de tip Leaf fără modificarea ierarhiei.

**Exemple de ierarhii recursive:**

- Grup de elemente grafice care pot fi translatate, scalate, etc. Un grup poate conține alte grupuri.
- Ierarhii de fişiere şi foldere. Un folder poate contine fişiere şi alte foldere. Aceleaşi operatii pot fi folosite pentru redenumire, mutare sau ştergere a fisierelor sau a folderelor.

# Șablonul Composite (3)




Fiecare frunză implementează interfața *Component*.

Agregatele Panel si Window specializează clasa abstractă *Composite*.

Mutarea sau redimensionarea unui obiect Composite are impact asupra tuturor componentelor sale.

# Șablonul Observer (1)

**Nume:** Observer, Subject-Observer, Dependents, Publish-Subscribe; **tipul**: comportamental

**Descrierea problemei**: definirea unei dependențe între obiecte, astfel încât atunci când un obiect își schimbă starea toate obiectele dependente de el să fie notificate şi actualizate.

**Motivația:** cresterea gradului de reutilizare a claselor, prin separarea obiectelor dependente în clase distincte slab cuplate.

**Exemple**:

1. In multe aplicații se doreste separarea datelor (si logicii) aplicației de prezentarile lor in interfața utilizator. In acest fel, **clasele care definesc datele şi logica aplicației şi cele care realizează prezentarea/prezentările pot fi reutilizate independent**.

   Obiectele care realizeaza prezentarile sunt dependente de obiectul care contine datele.

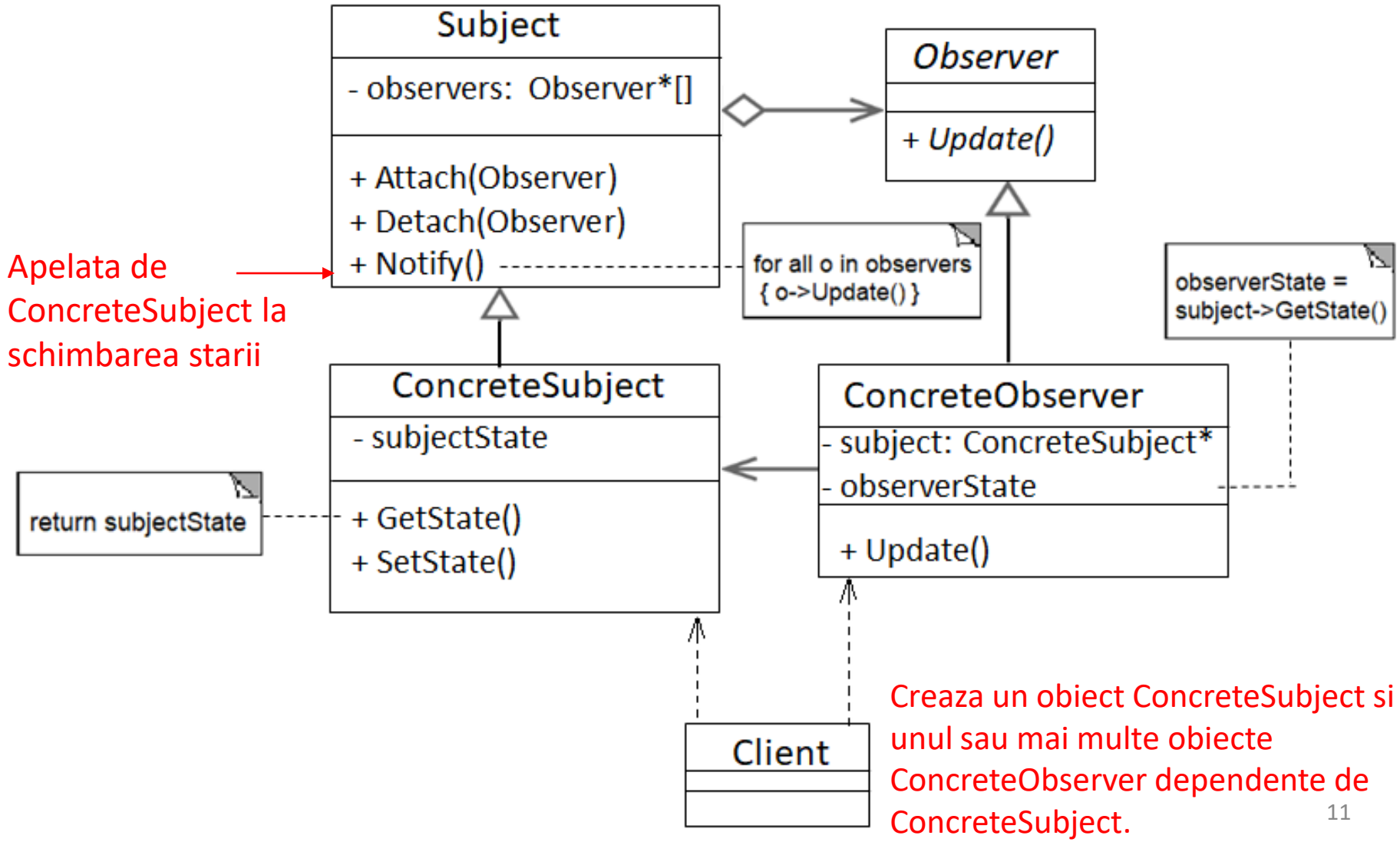
2. O aplicație care gestionează un magazin doreste să trimită notificări clienților care s-au abonat, de fiecare dată când se primeşte un produs, dar numai clienților care sunt interesați de acel produs.

   Clientii interesati de un produs sunt dependenti de continutul magazinului.

   Ei sunt notificati la aparitia produsului.

# Șablonul Observer (2)

**Soluția**



# Șablonul Observer (3)

**Soluția - continuare**

- Subiectul mentine o lista de observatori – obiectele dependente de subiect – pe care-i notifica ori de cate ori se modifica starea subiectului.

**Subject**

—Furnizează o interfață pentru ataşarea, detaşarea si notificarea observatorilor săi.

**Observer**

—Defineşte o interfață pentru actualizarea observatorilor.

**ConcreteSubject**

—Memorează starea de interes pentru obiectele **ConcreteObserver**.

—Trimite o notificare observatorilor săi atunci cand i se schimba starea.

**ConcreteObserver**

—Mentine o referinta la un obiect **ConcreteSubject**.

—Implementează funcția Update() pentru  a păstra starea sa consistentă cu a subiectului.

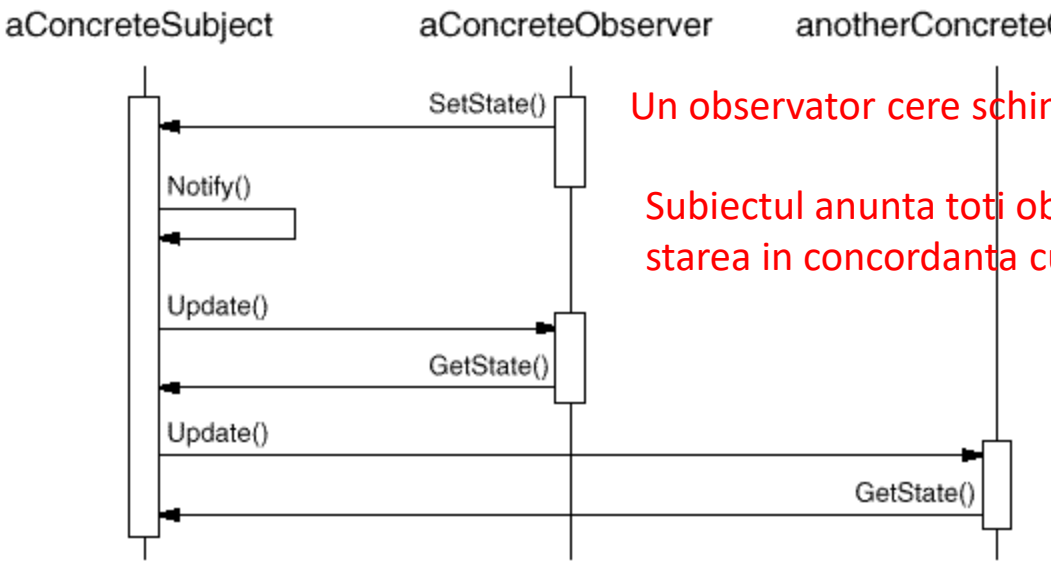
# Șablonul Observer (4)

- Obiectele cheie in acest sablon sunt **subiectul** – obiectul independent (Subject, Publisher) si **observatorii –** obiectele dependente (Observers, Subscribers), care sunt notificate de subiect atunci cand anumite evenimente au produs schimbarea stării sale.
- Un subiect poate avea orice numar de observatori dependenti.
- Observatorii sunt notificati ori de cate ori subiectul isi schimba starea.
- Ca raspuns la notificare, fiecare observator va interoga subiectul pentru a-si sincroniza starea cu starea subiectului.
- Este posibil ca fiecare observator sa se înregistreze numai pentru anumite evenimente (schimbari de stare) produse la subiect, caz în care el va fi notificat numai la producerea acelor evenimente.
- Un observator se poate înregistra la mai multi subiecti.

# Șablonul Observer (5)

**Colaborări**

- Un obiect *ConcreteSubject* notifica observatorii sai ori de cate ori are loc o schimbare care ar putea sa faca starea observatorilor inconsistenta cu starea sa.
- Dupa ce a fost informat de o schimbare la *ConcreteSubject*, un obiect *ConcreteObserver* il poate interoga pentru a obtine informatii.
- Obiectul *ConcreteObserver* utilizeaza aceste informatii pentru a-si adapta starea cu aceea a obiectului *ConcreteSubject.*




Un observator cere schimbarea starii subiectului

Subiectul anunta toti observatorii ca trebuie sa-si actualizeze starea in concordanta cu noua sa stare: Udate()

Pentru actualizarea starii, fiecare observator cere noua stare a subiectului: GetState()

# Șablonul Observer (6)

**Consecințe**

- Sablonul Observer permite definirea independenta a subiectilor si a observatorilor. **Subiectii si observatorii pot fi reutilizati independent unii de altii**. **Pot fi adaugati noi observatori fara a modifica subiectul sau alti observatori**.

Alte aspecte ale utilizarii sablonului Observer sunt:

- ***Cuplarea abstracta între Subject şi Observer***.

  Subiectul are doar o lista de observatori care expun interfata simpla a clasei abstracte Observer. Subiectul nu cunoaste clasa concreta a niciunui observator. In acest fel, **cuplarea intre subiecti si observatori este abstracta si minimala**. Deoarece subiectul si observatorul nu sunt puternic cuplati, ei pot apartine la diferite nivele de abstractizare ale unui sistem.

# Șablonul Observer (7)

- ***Suport pentru o comunicare broadcast.***

  Atunci cand i se schimba starea, subiectul trimite o notificare catre toate obiectele observator care s-au inregistrat la el. Subiectul nu trebuie sa stie cate obiecte interesate exista; singura sa responsabilitate este sa notifice observatorii sai. In acest fel, observatorii pot fi adaugati sau eliminati in orice moment. Este datoria observatorului sa trateze sau nu o notificare.

- ***Actualizari neasteptate.***

  Deoarece observatorii nu stiu unul despre altul ei nu pot cunoaste costul unei modificari la subiect. Astfel o foarte mica modificare la subiect poate antrena un numar mare de actualizari la observatori si obiectele dependente de ei. Problema este agravata de faptul ca protocolul foarte simplu de actualizare nu da detalii despre ce anume s-a schimbat la subiect.

# Şablonul Observer (8)
## Exemple de implementare în C++

**Exemplul 1**

```
#include <list>
using namespace std;
class Observer;

class Subject
{ public:
      virtual ~Subject();
      virtual void Attach(Observer*);
      virtual void Detach(Observer*);
      virtual void Notify();
 protected:
      Subject();
 private:
      list<Observer*> observers;
};

 void Subject::Attach (Observer* o)
 { observers.push_back(o); }

 void Subject::Detach(Observer*o)
 { observers.remove(o); }
```

Interfața **Observer** este definita printr-o clasa abstracta:

```
class Observer
{     public:
          virtual ~ Observer();
           virtual void Update(
                          Subject* theChangedSubject) = 0;
      protected:   Observer();
};
```

Implementarea admite ca un observator sa aiba mai multi subiecți.

Subiectul transmis operatiei **Update()** permite observatorului să determine care subiect şi-a schimbat starea atunci cand el observă mai multi subiecți.

# Șablonul Observer (9)

```
void Subject::Notify ()
 {
     for (auto it=observers.begin(); it!=observers.end(); it++)
                        (*it)->Update(this);
}
```

**ClockTimer** este un ***subiect concret*** pentru memorarea si actualizarea orei curente. El notifica observatorii la fiecare secunda.

```
class ClockTimer : public Subject
{  public:  ClockTimer();
          virtual int GetHour();
          virtual int GetMinute();
          virtual int GetSecond();
          void Tick();
  protected:
       int ora, minutul, secunda; // starea interna
}
void ClockTimer::Tick() // apelata de un timer intern al calculatorului, la intervale regulate
{ * actualizeaza starea interna a subiectului concret
   Notify(); // anunta observatorii despre schimbarea starii subiectului
 }
```

# Şablonul Observer (10)

Sunt definite doua clase “ConcreteObserver”: DigitalClock si AnalogClock. Ambele implementeaza interfata **Observer** şi afiseaza timpul curent memorat în clasa ClockTimer.

Pentru afişarea timpului se utilizeaza o clasa a interfetei grafice, numita Widget:

```
class Widget
{ public:
  virtual void Draw();
};

class DigitalClock: public Widget, public Observer
{    public: DigitalClock(ClockTimer*); //pointer la subiectul concret
        virtual ~DigitalClock();
        virtual void Update(Subject*); // implementeaza operatia Update() a Observer-ului
        virtual void Draw();  // redefineste operatia clasei Widget: defineste modul de afisare
                              //a ceasului digital;
    private:
            ClockTimer* subject; // pointer la “subiectul concret”
 };
```

# Șablonul Observer (11)

```
 DigitalClock::DigitalClock (ClockTimer* s)
{   subject = s;
    subject->Attach(this); // se ataseaza la subiect
 }
DigitalClock::~ DigitalClock ()
 {  subject->Detach(this);
 }
void DigitalClock::Update (Subject* theChangedSubject)
{  if (theChangedSubject == subject)
      Draw();
}
 void DigitalClock::Draw ()
{   // preia starea curenta a subiectului
   int hour = subject->GetHour();
   int minute = subject->GetMinute();
   // etc.
  *draw the digital clock
  }
```

# Şablonul Observer (12)

In mod similar este definita o clasa AnalogClock.

```
class AnalogClock : public Widget, public Observer
{  public:  AnalogClock(ClockTimer*);
   virtual void Update(Subject*);
   virtual void Draw();
    // ...
};
```

**Client**

// Creaza un obiect AnalogClock şi unul DigitalClock care vor afişa intotdeauna acelasi timp:

```
ClockTimer* timer = new ClockTimer;    //subiectul concret
AnalogClock* analogClock = new AnalogClock(timer);
DigitalClock* digitalClock = new DigitalClock(timer);
```

# Șablonul Observer (13)

**Exemplul 2**

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

class Observer;

class Subject
{    public:
       ~Subject() {}
       virtual void Attach(Observer* o)
           { observers.push_back(o); }
       virtual void Detach(Observer* o)
           { observers.remove(o); }
       virtual void Notify()
       {    for (auto it=observers.begin(); it!=observers.end(); it++)
              (*it)->Update(this);
       }
    protected:
       Subject() {}
    private:
       list<Observer*> observers;
};
```

```
class Observer
{  public:
       virtual ~ Observer() {}
       virtual void Update(Subject* theChangedSubject) = 0;
    protected:
       Observer() {}
};
```

# Șablonul Observer (14)

```
#include <iostream>
using namespace std;
#include <math.h>

class SubjectX : public Subject
{
   int x;
   public:
    SubjectX (int _x=0) {}
    void SetX (int _x)
    {
      x=_x;
      Notify();
    }
    int GetX ()
    {  return x; }
};

class Observer2 : public Observer
{
   public: void Update(Subject * s)   { cout << pow( ((SubjectX*)s)->GetX(), 2 ) << endl; }
};
class Observer3 : public Observer
{
   public: void Update(Subject * s)   { cout << pow( ((SubjectX*)s)->GetX(), 3 ) << endl; }
};

int main()
{  Observer2 o2;
   Observer3 o3;

   SubjectX sx;
   sx.Attach(&o2);
   sx.Attach(&o3);

   SubjectX sxi;
   sxi.Attach(&o2);

   sx.SetX(5);
   sxi.SetX(10);

   return 0;
}
```

*Output:*

25
125
100

# Lecturi suplimentare


1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns - Elements of Reusable  Object-Oriented Software, cunoscuta si sub numele “Gang of Four” (GoF), Addison Wesley,1994.
2. https://sourcemaking.com/design_patterns
3. https://refactoring.guru/design-patterns
4. https://www.tutorialspoint.com/design_pattern