Design Patterns

Ideea de utilizare de patternuri în proiectare fost introdusă de un arhitect Christopher Alexander care a dat şi definţia lor: "*Fiecare şablon descrie o problemă care apare mereu în domeniul nostru de activitate şi indică esenţa soluţiei acelei probleme, într-un mod care permite utilizarea soluţiei de nenumărate ori în contexte diferite*".

Şabloanele de proiectare se utilizeaza în toate domeniile care implică o activitate de proiectare: constructii, aeronave, etc.
În domeniul sistemelor soft (O-O) soluţiile sunt exprimate în termeni de obiecte şi interfeţe (în loc de ziduri, uşi, grinzi, came etc); esenţa noţiunii de şablon este aceeaşi: *Un pattern (şablon) reprezintă o soluţie comună a unei probleme într-un anumit context*.

Clasificare
Şabloanele utilizate în sistemele OO :

- idiomuri
- mecanisme
- cadre (frameworks).

Un ***idiom*** este legat de un anumit limbaj de programare şi se referă la practici sau obiceiuri "bune", care se indica a se utiliza când utilizăm respectivul limbaj.

Un ***mecanism*** (mechanism) este o structură în cadrul căreia obiectele colaborează în vederea obţinerii unui anumit comportament care satisface o anumită cerinţă a problemei. Mecanismele reprezintă decizii de proiectare privind modul în care cooperează colecţiile de obiecte. Ele se mai numesc **şabloane de proiectare** (***design patterns***).

Un **cadru** (framework) descrie şi menţine cu ajutorul unui set de clase abstracte relaţii între obiecte . De exemplu, un editor grafic este specializat pentru diferite domenii: editor muzical, editor CAD, etc. Scheletul unei aplicaţii "editor-grafic" (arhitectura de nivel înalt) poate fi gândit fără să ţin cont de domeniul particular în care se va folosi. Indiferent de tipul editorului, se pot "inventa" un set de clase abstracte, relaţii intre ele şi cod generic ce va fi reutilizat în fiecare caz concret (vezi exemplu de la Factory Method)

Şabloanele de proiectare

Un proiectant experimentat ştie ca NU TREBUIE să rezolve fiecare problemă începând de la zero, ci reutilizând soluţii (bune) din proiecte anterioare. Atunci când descoperă o soluţie bună o va folosi mereu.

Pentru descrierea pattern-urilor se utilizează notaţii grafice (UML) şi un limbaj care permite o descriere uniformă a tuturor pattern-urilor. Formatul de descriere variază de la autor la autor, dar în general cuprinde următoarele secţiuni:

- **numele** şablonului: descrie sintetic problema rezolvată de şablon şi soluţia; când este un pattern "clasic" se precizează şi categoria din care face parte.
- **scop (Intent)** : pe scurt, ce problemă rezolvă
- **motivaţia**: un exemplu adecvat de problemă rezolvată
- **problema**: o descriere mai largă a problemei rezolvate şi a contextului în care ea apare;

- **soluția**: o descriere a elementelor de proiectare utilizate şi a relațiilor dintre ele. Soluția nu descrie un proiect particular sau o implementare concretă, ci un ansamblu abstract de clase şi obiecte care rezolvă un anumit gen de probleme de proiectare;
- **consecințele (bune şi rele)** implicate de folosirea şablonului: acestea pot privi impactul asupra flexibilității, extensibilității sau portabilitătii sistemului, după cum pot să se refere la aspecte ale implementării sau limbajului de programare utilizat. Compromisurile sunt de cele mai multe ori legate de spațiu şi timp.

Criterii de clasificare:

scop: şabloanele pot fi, din acest punct de vedere: creaționale, structurale sau comportamentale.

- Şabloanele **creaționale** (creational patterns) privesc modul de creare a obiectelor.
- Şabloanele **structurale** (structural patterns) se referă la compoziția claselor sau a obiectelor.
- Şabloanele **comportamentale** (behavioral patterns) caracterizează modul în care obiectele şi clasele interacționează şi îşi distribuie responsabilitățile.

domeniu de aplicare: şabloanele se pot aplica obiectelor sau claselor.

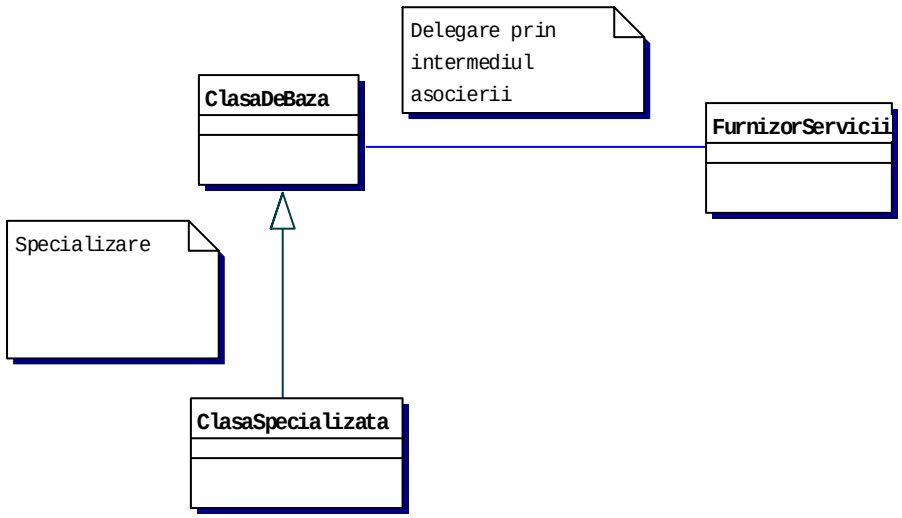
- Saloanele pentru **obiecte** se referă la relațiile dintre instanțe, relații care au un caracter dinamic.
  - Şabloanele creaționale ale obiectelor acoperă situațiile în care o parte din procesul creării unui obiect cade în sarcina unui alt obiect.
  - Şabloanele structurale ale obiectelor descriu căile prin care se asamblează obiecte.
  - Şabloanele comportamentale ale obiectelor descriu modul în care un grup de obiecte cooperează pentru a îndeplini o sarcină ce nu ar putea fi efectuată de un singur obiect.
- Şabloanele **claselor** se refera la relațiile dintre clase, relații stabilite prin moştenire şi care sunt statice (fixate la compilare).
  - Şabloanele creaționale ale claselor acoperă situațiile în care o parte din procesul creării unui obiect cade în sarcina subclaselor.
  - Şabloanele structurale ale claselor descriu modul de utilizare a moştenirii în scopul compunerii claselor.
  - Şabloanele comportamentale ale claselor utilizează moştenirea pentru descrierea unor algoritmi şi fluxuri de control.

Scop-Domeniu de aplicare	Creationale	Structurale	Comportamentale
Clasa	Factory Method	Adapter (class)	Interpreter Template Method
Obiect	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Observații generale legate de OOP

Moştenirea de clasă se caracterizează prin următoarele:

- este definită static, la compilare, şi poate fi specificată direct, fiind suportată explicit de limbajele de programare;
- permite modificarea uşoară a implementării operațiilor reutilizate, şi anume, într-o subclasă care redefineşte o parte din operațiile clasei părinte pot fi afectate şi operații moştenite, dacă acestea apelează operații redefinite.
- implementarea moştenită de la clasele părinte nu poate fi modificată la momentul execuției;
- cel mai adesea clasele părinte definesc cel puțin parțial reprezentarea fizică a subclaselor lor, deci subclasele au acces la detalii ale implementării superclaselor. De aceea se mai spune că moştenirea de clasă încalcă principiile încapsulării;
- modificările aduse implementării unei superclase vor forța subclasele să se modifice şi ele. Dependențele de implementare pot cauza probleme atunci când se încearcă reutilizarea subclaselor: dacă anumite aspecte ale implementării moştenite nu corespund necesităților aplicației clasa părinte trebuie rescrisă sau înlocuită. Această dependență limitează flexibilitatea şi, în ultima instanță, reutilizarea. O soluție în acest caz ar fi aplicarea moştenirii de la clase abstracte, deoarece ele includ implementare în mica măsură.



Compunerea obiectelor se caracterizează prin:

- se defineşte în mod dinamic, la execuție, prin faptul că anumite obiecte primesc referințe ale altor obiecte;
- necesită ca obiectele să-şi respecte unul altuia interfața, ceea ce presupune ca interfețele să fie proiectate astfel încât să nu împiedice utilizarea unui obiect în combinație cu mai multe tipuri de obiecte. Deoarece obiectele sunt accesate doar prin intermediul interfețelor, nu este încălcat principiul încapsulării. În decursul execuției orice obiect

poate fi înlocuit cu altul, atâta timp cât obiectele respective au acelaşi tip. În plus, datorită faptului că şi implementarea unui obiect este scrisă tot în termenii interfeţelor altor obiecte, dependenţele de implementare vor fi substanţial reduse;

- prin compunerea obiectelor se obţin următoarele efecte asupra unui proiect: clasele sunt încapsulate şi "concentrate" asupra câte unui singur obiectiv, ceea ce face ca ele, ca şi ierarhiile lor, să aibă dimensiuni mici şi să fie mai uşor de gestionat. Un proiect bazat pe compunerea obiectelor se caracterizează printr-un număr mai mare de obiecte şi un număr mai mic de clase, iar comportarea sistemului va depinde de relaţiile dintre obiecte, în loc sa fie definită de o anumită clasă.

Experienţa arată că adesea proiectanţii folosesc moştenirea în mod abuziv. De aceea se recomandă studiul şi aplicarea şabloanelor de proiectare, acestea bazându-se foarte mult pe compunerea obiectelor.

Delegarea

Reprezintă o cale de aplicare a principiului compunerii obiectelor. Intr-o relaţie de delegare 2 obiecte sunt implicate în rezolvarea unei cereri, şi anume: obiectul care receptează mesajul (delegatorul) deleagă execuţia operaţiei corespunzătoare unui alt obiect - delegat.

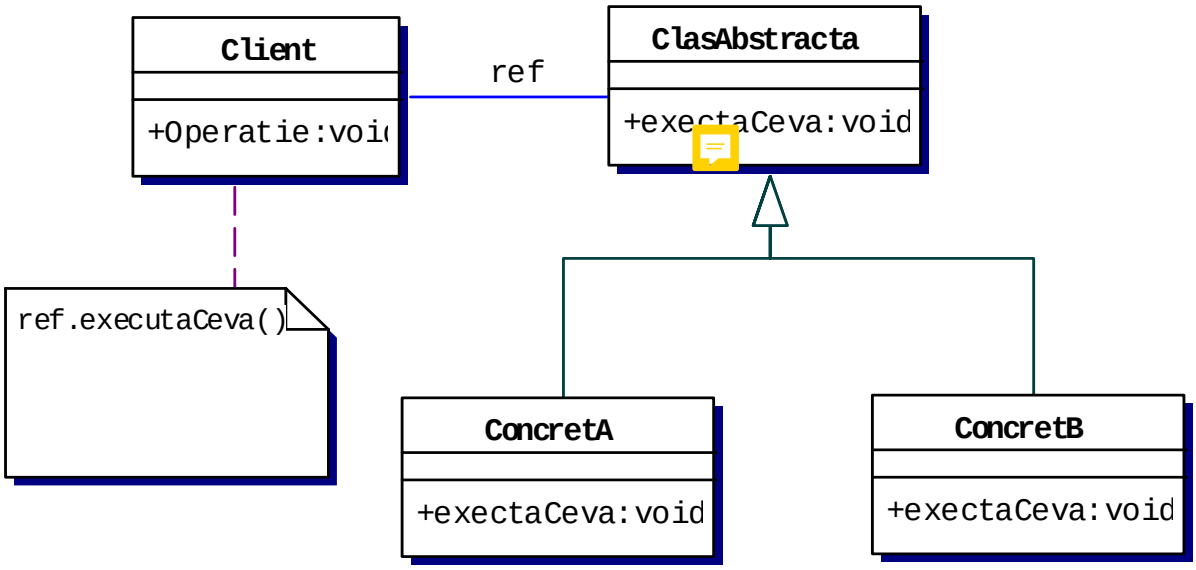


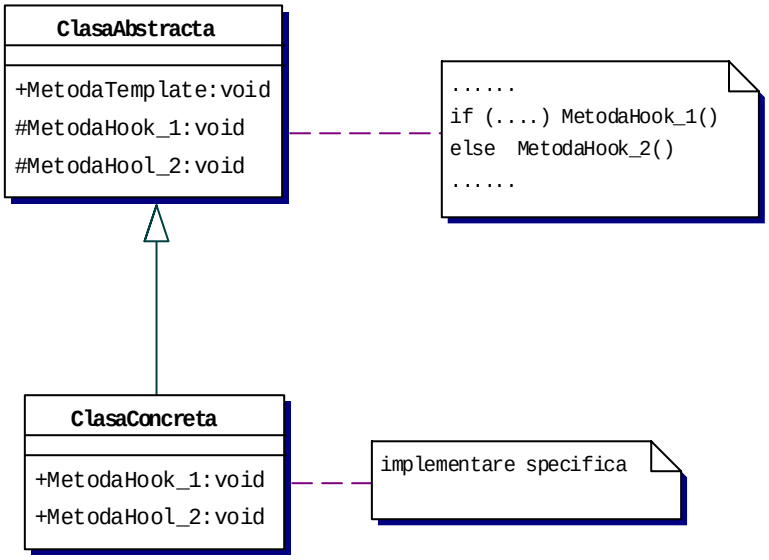
Diagrama de mai sus a fost propusă de W. Zimmer şi este conoscută ca "*Objectifier Pattern*"

Acest lucru este oarecum similar cu situaţia în care subclasele "pasează" sarcina execuţiei unor operaţii claselor părinte (este vorba despre operaţiile moştenite şi neredefinite). Dar, în timp ce clasa părinte a unei subclase rămâne aceaşi pe toată durata execuţiei, în cazul delegării, *obiectele delegat pot fi schimbate*, cu condiţia să aibă aceaşi interfaţă.

Delegarea este considerată ca un şablon de proiectare fundamental, pe ea bazându-se foarte multe din celelalte şabloane (ex: State, Visitor, Strategy, Mediator, Chain of Responsability, Bridge).

Reguli care favorizează flexibilitatea şi extensibilitatea în sistemele OO

- Clasele au scop limitat
- Metodele să fie scurte şi simple
- Când avem de ales, preferăm delegarea, specializării (moştenirii)
- Se utilizează pattern-ul Template



- **Problema** :

```
class SomeClass {
  ..
  void computeResult () {
    x = ..;
    y = f(x);
    ..
    result = y / a;
  }
}
class SomeModifiedClass extends   SomeClass {
  ..
  void computeResult () {
    x = ..; // copie (redundant)
    y = f(x); // -"-
    .. // -"-
    result = y / (a + b); // diferit
  }
}
```

- **Template**

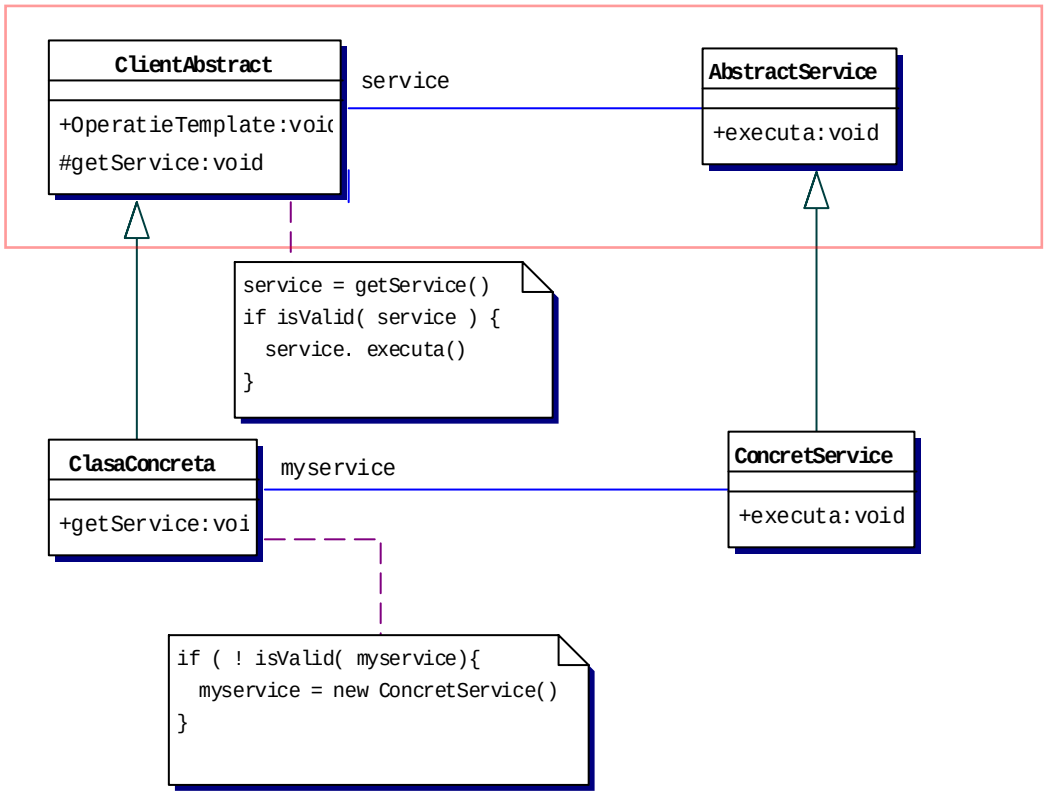
```
class SomeClass {
  ..
  void computeResult () { // schelet
    x = ..;
    y = f(x);
```

```
  ..
  result = g(y);
 }
 double g ( double y ) { // operation
   return y / a;
 }
}

class SomeModifiedClass extends SomeClass {
 ..
 double g ( double y ) { // surcharge...
   return y / (a + b);
 }
}
```

Combinarea pattern-urilor

Combinand cele două pattern-uri (Objectifier şi Template), obținem: “The Flexible Service Pattern”



- utilizăm Objectifier pentru a crea o ierarhie de servicii (dreapta). Serviciile concrete au interfața impusa de AbstractService
- Metode de genul GetService() sunt metode “hook” de la Template. Rescrierea lor în clasele corespunzătoare clienților oferea o posibilitate flexibila de a alege un serviciu concret